Cursus C en C++

Leo Rutten

3/12/2024

Deel C

Voorwoord

Voor het academiejaar 2017-2018 is deze tekst de cursus voor het vak C/C++. De tekst bestaat hoofdzakelijk uit de delen C en C++ die voorheen al geruime tijd als aparte delen bestonden. Recent zijn deel 3 over STL, deel 4 over C++11/C++14 en deel 5 over Qt bijgevoegd.

Vermits het vak niet meer kan steunen op eerdere vakken waarin C onderwezen werd, moet dit vak starten met een kennismaking met C. De studenten die dit vak volgen, hebben wel een eerste ervaring met Java en kennen bijgevolg de controlestructuren van de `if`, `for` en `while` die overeenkomen met C maar begrippen zoals pointers en arrays zijn ongekend of fundamenteel anders dan Java. Een goed begrip van de werking van pointers is essentieel om met succes C en zeker ook C++ programma's te kunnen ontwerpen. We zouden wel graag leven in een wereld waarin pointers en pointerfouten niet voorkomen maar zolang beide programmeertalen nog gebruikt worden, is een goede kennis hiervan noodzakelijk.

Deze cursus maakt een opsplitsing tussen de talen C en C++. Dit is te verklaren door het feit dat in het verleden beide tekstdelen als aparte cursussen hebben bestaan. De tekst over C++ is later ontstaan dan de tekst over C. Dit komt overeen met de ontstaansgeschiedenis van C en C++.

Hier zijn enkele argumenten om C en C++ apart te behandelen:

- C is ouder dan C++. Er is een periode geweest dat C wel en C++ niet bestond. In deze periode werd C veelvuldig gebruikt en was het een levende programmeertaal en is het nog steeds.
- C wordt nog steeds gebruikt. Er zijn nog steeds omgevingen waar geen C++ maar wel C wordt gebruikt. Dit zijn dikwijls eenvoudige platformen zoals microcontrollers die niet krachtig genoeg zijn om C++ te ondersteunen. Meestal is er in die omgevingen geen heap en kan je dan ook geen dynamisch objecten maken zoals we in C++ en Java gewoon zijn. In deze beperkte omgevingen kan de taal C bloeien: toch gebruik maken van een hogere programmeertaal en tegelijkertijd relatief snelle programma's schrijven.
- Er zijn nog altijd aparte compilers voor C en C++. Meestal is het wel zo dat dezelfde compiler beide talen kan compileren maar je moet altijd wel met een optie aangeven welke syntax (C of C++) er gebruikt wordt. Maar er zijn ook situaties waar de compiler alleen maar C verstaat. Dan mag je geen constructies gebruiken die alleen in C++ voorkomen.
- C en C++ zijn aparte talen, dus ook de standaardisatie. De laatste goedgekeurde standaard van C is `C11` en die van C++ is `C++14`[1]. Beide standaarden zijn recent[2].
- Als je de verschillen kent tussen C en C++, kan je je ook gemakkelijker een oordeel vormen over programmeertalen en de evolutie ervan.

In deel I wordt de programmeertaal C behandeld. Dit is een imperatieve niet-objectgeoriënteerde taal. Wie zijn programmeercarrière start met een taal zoals Java, Python of Ruby zal natuurlijk opwerpen dat de taal C een stap terug is: van objectgeoriënteerd terug naar niet-objectgeoriënteerd. Dit argument is correct maar kan meteen aangevuld worden met een extra argument. C++, wel objectgeoriënteerd,

[1] De C++14 standaard is goedgekeurd in december 2014 en de goedkeuring van `C++17` is gepland voor juli 2017. En ondertussen wordt C++20 ook al gepland. Tot over 3 jaar!

[2] De C++14 standaard is momenteel goed ondersteund in de huidige versie van de GNU `g++` compiler. Het is deze versie die we in het labo zullen gebruiken.

gebruikt grotendeels concepten die nog stammen uit de eerste dagen van de taal C. Het gebruik van de controlestructuren, de pointers en het bijbehorende geheugenmodel is hetzelfde gebleven bij de overgang van C naar C++. Vandaar dat het belangrijk is om toch zeker een goed inzicht te verwerven van de werking van C en vooral van de pointers.

Wie dit niet door heeft, zou in zijn programma's regels kunnen schrijven zoals de volgende:

```
int *p;
*p = 17;
```

Je ziet hier meteen dat er sprake is van een niet-geïnitialiseerde pointer en dat bijgevolg een toekenning naar de geheugenruimte die door deze pointer wordt aangewezen, fataal is voor het programma. Het correct gebruik van pointers is essentieel voor C maar ook voor C++. Daarom is dit het belangrijkste argument om het deel over C op te nemen in deze cursustekst.

Niet alle hoofdstukken van deel 1 zijn essentieel: vanaf het hoofdstuk over de `struct` worden technieken zoals gelinkte lijsten gebruikt die allang kunnen vervangen worden door de STL bibliotheek van C++. Voor de volledigheid van de tekst over C zijn deze hoofdstukken toch opgenomen in deel 1.

Kennismaking met C

De programmeertaal C vindt zijn oorsprong bij het ontstaan van het operating systeem UNIX. De eerste versie van UNIX was in assembler geschreven. Om de overdracht naar computersystemen met een andere processorarchitectuur mogelijk te maken hebben de auteurs de hogere programmeertaal C ontworpen. Deze taal kreeg een aantal eigenschappen die interessant zijn voor het ontwerp van systeemsoftware. Deze kenmerken zijn van assembler overgenomen: directe toegang tot de hardware en datatypen die op maat gemaakt zijn van de interne registers. Deze goede eigenschappen hebben ertoe bijgedragen dat de taal C nog steeds een grote populariteit kent in de computerwereld. Voor elke processor is er tegenwoordig een C compiler. Dankzij goede en goedkope compilers is de taal C ook beschikbaar voor microcontrollerarchitecturen.

C staat sterk in domeinen waar ofwel snelheid ofwel overdraagbaarheid een belangrijke eis is. Vermits C een taal is waar je nog de band met assembler kan voelen, is het ook een taal die geliefd is om bepaalde programma's te ontwerpen waar snelheid het belangrijkste criterium is. Een ander eigenschap waar C goed in scoort, is de overdraagbaarheid. C programma's kunnen zodanig geschreven worden dat ze zonder wijziging kunnen gehercompileerd worden voor verschillende processorarchitecturen. Een goed voorbeeld hiervan is de Linux kernel. Deze draait zowat op elk platform.

Geschiedenis

Dit zijn enkele markante punten uit de geschiedenis van C:

1976 De BCPL taal wordt ontworpen door Martin Richards.
1970 De eerste versie UNIX in B door Ken Thompson geschreven, verschijnt.
1972 De C taal wordt ontworpen voor versie UNIX op DEC PDP11 door Dennis Ritchie.
1978 Het boek 'The C programming language' door Brian Kernighan & Dennis Ritchie geschreven verschijnt bij Prentice Hall.
1982 ANSI start de standaardisatie van C.
1986 Een nieuwe uitgave van 'The C programming language' verschijnt.
1989 De ANSI C standaard verschijnt. In deze standaard worden de functieprototypes ingevoerd.
1999 De C99 standaard verschijnt.
2011 De C11 standaard verschijnt. Deze standaard zorgt voor een verbeterde compatibiliteit met C++.

De taal C is dus ontstaan bij het ontwerp van UNIX. Tot dan toe schreef men de programma's van operating systemen in assembler. Door in C te programmeren is UNIX overdraagbaar op elk type computer dat een C compiler kent.

Dit zijn de kenmerken van C:

- algemeen
- flexibel
- overdraagbaar

- laat programmeren op laag niveau toe
- kan dus assembler vervangen
- gebaseerd op types
- veel verspreid
- niet voor beginners

Eerste voorbeelden

Als kennismaking met de taal C beginnen we met enkele eenvoudige programma's. Ze zijn bijna zonder verdere uitleg te begrijpen.

We starten met een eerste voorbeeld:

```
#include <stdio.h>

int main() // een eenvoudig programma
{
   int num;

   num = 1;
   printf("dit is");
   printf(" een eenvoudig programma\n");
   printf("%d is het eerste gehele getal\n",  num );

   return 0;
}
```

Dit programma zet de volgende tekst op het scherm:

```
dit is een eenvoudig programma
1 is het eerste gehele getal
```

Als we dit programma nalezen, vinden we notaties die specifiek zijn voor C. De regel `#include <stdio.h>` geeft aan dat een ander bestand in dit programmabestand tussengevoegd wordt. De naam van het bestand is `stdio.h` en bevat definities die nodig zijn bij de standaard in- en uitvoer (`stdio` is de afkorting van standard input output). Deze definities hebben betrekking op de invoer van het toetsenbord en de uitvoer naar het scherm.

Op de volgende regel treffen we `main()` aan. Hiermee geven we aan dat dit programma bestaat uit een functie die main heet. Een functie is steeds te herkennen aan de 2 ronde haken achter de naam. Een C programma is opgebouwd uit functies die elk bepaalde taken uitvoeren. Wanneer het programma start, wordt steeds de functie `main()` gestart. Met behulp van de tekens `{` en `}` worden opdrachten gegroepeerd bij een functienaam. Verschillende opdrachten tussen accolades noemt men een blok of een samengestelde opdracht.

Achter de naam `main` treffen we commentaar aan. Dit wordt aangegeven door de symbolen `/*` en `*/`. De begin- en eindeaanduiding moeten niet op dezelfde regel staan. Sommige compilers laten ook commentaar binnen commentaar toe. Dit kan handig zijn als we bijvoorbeeld een stuk programma in commentaar zetten om de uitvoering ervan tijdelijk over te slaan. Je kan commentaar voor één regel ook starten met `//`.

Als eerste opdracht in dit blok treffen we een declaratie aan.

```
int num;
```

Hiermee wordt aangegeven dat het programma gebruik maakt van een variabele met de naam `num`. Deze variabele kan gehele getallen opslaan.

Variabeledeclaraties worden steeds aan het begin[3] van een blok vermeld. Hiermee geven we aan dat een naam verder in het programma als variabele dienst doet en dat die variabele één waarde van een bepaald type kan opslaan.

[3] In C++ hoeft dat niet meer het begin te zijn. In het midden van een blok mag ook. Vanaf de standaard C99 is dit ook zo.

Met een toekenning wordt een waarde in een variabele geplaatst.

```
num = 1;
```

Deze toekenningsopdracht plaatst de waarde 1 in de variabele.

De volgende opdracht die we aantreffen, doet schermuitvoer[4]. `printf()` is één van de standaard in- en uitgave functies en plaatst een tekst op het scherm. De tekst plaatsen we tussen de symbolen `" "`. In C noemen we een tekst tussen dubbele aanhalingstekens een string. In een string kunnen we alle leesbare tekens plaatsen. Tekens met een speciale betekenis worden door de backslash voorafgegaan. Bijvoorbeeld: `\n`, dit teken doet de cursor naar het begin van de volgende regel verplaatsen.

Met behulp van `%d` wordt duidelijk gemaakt dat een geheel getal in de tekst ingelast moet worden. De variabelenaam wordt na de string vermeld. Het `%` teken gevolgd door een letter specificeert het formaat.

We geven een tweede voorbeeld waarin een berekening voorkomt:

```
#include <stdio.h>

int main()
{
   int voet,vadem;

   vadem = 2;
   voet = 6 * vadem;
   printf("in %d vadem zijn er %d voet\n",vadem,voet);

   return 0;
}
```

Bij de tweede toekenning van dit programma zien we aan de rechterzijde een uitdrukking. We kunnen niet alleen een waarde toekennen aan een variabele, maar ook het resultaat van een uitdrukking. In de tekst die door `printf` op het scherm wordt gezet, komen nu 2 getallen voor. De eerste `%d` wordt vervangen door de inhoud van vadem, de tweede `%d` door de inhoud van voet. Elke percentaanduiding komt overeen met een waarde die na de string vermeld wordt. Het aantal waarden moet precies overeenkomen met het aantal percentaanduidingen.

En nog een voorbeeld:

```
#include <stdio.h>

void help()
{
   printf("hier is hulp\n");
}

int main()
{
   printf("ik heb hulp nodig\n");
   help();
   printf("dank u");

   return 0;
}
```

In dit programma treffen we 2 functies[5] aan: `main()` en `help()`. Het programma start per definitie met de uitvoering van `main()`. Deze functie doet eerst schermuitvoer met `printf()` en roept daarna de functie `help()` op. Dit betekent dat alle opdrachten van `help()` uitgevoerd worden. Daarna gaat het programma verder met de opdracht in `main()` die volgt na de oproep van `help()`.

[4] Om de schermuitvoer te kunnen zien heb je een console nodig, dat is het venster (`bash` of andere) waar je commando's kan intikken. De console was een hele tijd onpopulair maar is opnieuw in opmars.

[5] Hier zie je het verschil met Java en C++: functies hoeven niet binnen klassen te staan. Klassen bestaan niet in C.

De programmeertaal laat dus toe dat we een aantal opdrachten groeperen in een functie. Als we de functie met zijn naam oproepen dan worden zijn opdrachten uitgevoerd. We besparen schrijfwerk door opdrachten die veel voorkomen in een programma, onder te brengen in een functie.

Werken met gegevens

Bij de vorige voorbeelden hebben we telkens het type int gebruikt om de variabelen te declareren. Dit type slaat gehele getallen op. We zien nu een aantal programma's waar andere types worden gebruikt. De C programmeur moet voor elke variabele een type kiezen dat de gunstigste eigenschappen heeft voor de waarde die opgeslagen moet worden. Als we het type `float` voor berekeningen gebruiken, dan moeten we er rekening mee houden dat de rekentijden langer zijn en de nauwkeurigheid groter is dan bij het type `int`.

Een voorbeeld met verschillende types

Het volgende voorbeeld maakt gebruik van de types `float` en `char`.

```
/* omzetting gewicht in goudwaarde */
int main()
{
   float gewicht,waarde;
   char piep;

   piep = '\007';
   printf("geef uw gewicht in kg\n");
   scanf("%f", &gewicht);
   waarde = 415000 * gewicht; /* 1 kg goud = 415000 bef */
   printf("%cUw gewicht in goud is %10.2f waard.%c\n",
   piep,waarde,piep);

   return 0;
}
```

De variabelen `gewicht` en `waarde` zijn van het type `float`. Dit wil zeggen dat ze reële waarden kunnen opslaan. Een `float` type is hier nodig omdat we ook cijfers na de komma willen opslaan.

De variabele `piep` is van het type `char`. Hierin kunnen dus tekens opgeslagen worden. Door de toekenning krijgt piep de waarde '\007'. Dit is een teken met als code de waarde 7. Constanten van het type `char` worden steeds tussen ' ' geplaatst. We zien dat voor elk soort gegeven een ander type wordt gebruikt.

Dit programma werkt interactief. Het leest een getal van het toetsenbord, rekent hiermee en plaatst het resultaat op het scherm. Uitgave met `printf()` kennen we al. Ingave van het toetsenbord gebeurt met de functie `scanf()`. Bij de oproep van deze functie wordt ook een string doorgegeven net zoals bij `printf()`. Deze string gaat hier niet naar het scherm, maar geeft aan wat er ingelezen moet worden. In de string vinden we een `%f` terug. Hierdoor weet de functie `scanf()` dat een `float` waarde ingelezen moet worden. Na de string wordt er vermeld in welke variabele deze waarde terecht komt. Deze variabele wordt steeds voorafgegaan door &. Dit is de adresoperator die we later nog zullen zien.

In de `printf()` functie treffen we `%c` en `%10.2f` aan. De eerste aanduiding dient om een `char` variabele op het scherm te plaatsen. De tweede plaatst een `float` getal op het scherm met een breedte van 10 en 2 cijfers na de decimale punt. Elke `%` aanduiding heeft een corresponderende variabele.

Gegevens: variabelen en constanten

Met een computer kunnen we gegevens verwerken. Dit betekent dat een computer gegevens opslaat en daarna manipuleert. In een programma doen we dit met behulp van variabelen en constanten. Een constante kan toegekend worden aan een variabele en deze variabele kan in de loop van het programma gewijzigd worden. Een constante daarentegen kan niet gewijzigd worden.

De gegevens die in een C programma bijgehouden worden, zijn steeds van een welbepaald type. Een variabele wordt gedeclareerd en hierdoor weet de computer in welk formaat de informatie in die variabele opgeslagen wordt. De algemene vorm van een declaratie ziet als volgt uit:

typenaam variabelenaam;

In plaats van een enkele naam kunnen we ook meerdere namen bij een type plaatsten.

```
int a,b,c;
```

Variabelenamen bestaan uit maximum 63 letters en cijfers[6]. Er mogen enkel letters en cijfers (de underscore _ is een letter) in voorkomen en het eerste teken moet een letter zijn.

We zullen nu de verschillende basistypes bespreken.

Het `int` type

Dit type wordt gebruikt om gehele getallen op te slaan.

parse_error	geheugenruimte:4 bytes
	bereik: -2147483648 tot +2147483647

Deze waarden gelden voor 32 bit Linux.

Voorbeeld:

```
int regendagen;
int uren,minuten;

uren = 5;
scanf("%d",&minuten);
```

Bij de declaratie zelf kunnen de variabelen geïnitialiseerd worden.

```
int regendagen = 25;
int uren,minuten = 3;
```

`minuten` krijgt de waarde 3 in het laatste voorbeeld, `uren` wordt niet geïnitialiseerd. Het is wellicht beter om om elke declaratie op een aparte regel te plaatsten.

Integer constanten worden voorgesteld door groepjes cijfers.

parse_error	decimaal
parse_error	octaal
parse_error	hexadecimaal

Een geheel getal dat start met `0` is octaal. Een getal dat start met `0x` is hexadecimaal.

Het is mogelijk om bij de uitgave op scherm het getalstelsel te bepalen. De inhoud van een `int` variabele kan met een `%d`, `%o` of `%x` op het scherm geplaatst worden. Het resultaat is dan decimaal, octaal of hexadecimaal. In het voorbeeld wordt driemaal dezelfde waarde in een ander talstelsel op het scherm geplaatst.

```
int main()
{
   int x = 100;

   printf("dec = %d, octaal = %o, hex = %x\n",x,x,x);

   return 0;
}
```

[6] Om precies te zijn is in C99 de lengte 63 voor interne variabelen en 31 voor externe namen. En is de maximale regellengte 4095.

Door middel van adjectieven `short`, `long` en `unsigned` kan het bereik van het `int` type aangepast worden. Het `int` type komt gewoonlijk overeen met de registergrootte van de computer. Op een 32 bit besturingssysteem is dit gewoonlijk 32 bit. Met `short` wordt het bereik verkleind, met `long` vergroot en met `unsigned` verkrijgen we een type waarin enkel positieve waarden opgeslagen kunnen worden. Hierbij gelden de volgende beperkingen: `short` en `int` moeten minstens 16 bit groot zijn en `long` minstens 32 bit, bovendien mag een `short` niet groter zijn dan `int`, die op zijn beurt niet groter mag zijn dan een `long`.

In elk geval is het zo dat een `short` een bitbreedte heeft van 16 bit, `long` 32 bit. De bitbreedte van een `int` was vroeger 16 bit, nu is dit minstens 32 bit.

Op een 32 bit Linux levert de `gcc` compiler de volgende mogelijkheden:

`short int`, `short`:	geheugenruimte: 2 bytes bereik: -32768 tot +32767
`long int`, `int` of `long`:	geheugenruimte: 4 bytes bereik: -2147483648 tot +2147483647
`long long int` of `long long`:	geheugenruimte: 8 bytes bereik: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, van -2^{63} tot $2^{63} - 1$
parse_error	geheugenruimte: 2 bytes bereik:0 tot +65535
`unsigned long` of `unsigned int`:	geheugenruimte: 4 bytes bereik:0 tot +4294967295
`unsigned long long int` of `unsigned long long`:	geheugenruimte: 8 bytes bereik: 0 to 18,446,744,073,709,551,615 of tot $2^{64} - 1$

Gehele getallen die eindigen met de letter `L` zijn `long` constanten:

```
123L 045L 0x1234L
```

Dit achtervoegsel mag bij decimale, octale en hexadecimale constanten toegepast worden.

Hier is een overzicht van alle achtervoegsels die momenteel gangbaar zijn:

Geen type: `int`, `long int` of `long long int`

`L` type: `long int` of `long long int`

`LL` type: `long long int`

`U` type: `unsigned int`, `unsigned long int` of `unsigned long long int`

`UL` type: `unsigned long int` of `unsigned long long int`

`ULL` type: `unsigned long long int`

Bij schermuitgave moet precies aangeduid worden van welke soort elke variabele is en op welke wijze deze variabele op het scherm komt.

```
int main()
{
   unsigned un = 40000;
   long ln = 2000000000;
   unsigned long uln = 4000000000;

   printf("un: %u ln: %ld uln: %lu\n",un,ln,uln);

   return 0;
}
```

We zien hier nieuwe formaataanduidingen. `%u` gebruiken we bij tekenloze `int` variabelen en `%l` gebruiken we bij `long` variabelen. De combinatie `%lu` is voor tekenloze `long` variabelen.

Het volgende schema geeft een overzicht:

	parse_error	parse_error	parse_error	parse_error	parse_error
met teken	parse_error	parse_error	parse_error	parse_error	parse_error
zonder teken	parse_error	parse_error	parse_error	parse_error	parse_error

Indien niet de juiste percentaanduiding gebruikt wordt, komt er een onvoorspelbaar resultaat op het scherm. Dit komt omdat de compiler deze overeenkomst niet controleert. De programmeur moet er dus goed op letten dat er een juiste overeenkomst is tussen percentaanduiding en het type.

Het `char` type

Variabelen van het `char` type worden gebruikt voor de opslag van tekens. We zien hier een voorbeeld van een declaratie:

```
char letter,teken;
char cijfer;
```

Dit type variabele gebruikt 1 byte geheugen en hierin wordt het teken opgeslagen als een getal van -128 tot +127. Constanten van het `char` type worden tussen 2 aanhalingstekens genoteerd.

```
'A' 'c' '0'
```

Tekens met een speciale betekenis worden met een backslash voorgesteld.

`'\n'` nieuwe lijn

`'\t'` tab

`'\a'` belsignaal

`'\b'` backspace

`'\f'` formfeed

`'\r'` carriage return

`'\t'` horizontale tab

`'\v'` verticale tab

`'\\'` backslash

`'\?'` vraagteken

`'\''` enkel aanhalingsteken

`'\"'` dubbel aanhalingsteken

`'\0'` nulteken

We kunnen de code ook zelf samenstellen. Er wordt dan een getal achter de backslash geplaatst. Bijvoorbeeld `'\0'` is een `char` constante met 0 als code. Dit getal is octaal: `'\12'` heeft als waarde 10 decimaal. Als we de code in hexadecimaal willen uitdrukken, moet de letter `x` tussen de backslash en het getal geplaatst worden.

code in octaal: `'\ddd'`

```
'\123'
```

code in hexadecimaal: `'\xddd'`

```
'\x1b'
```

We kunnen deze speciale tekens ook in strings toepassen.

```
printf("\007wakker worden!\n");
```

In dit voorbeeld sturen we de code voor het belsignaal naar het scherm. Het teken `'\n'` plaatst de cursor op het begin van de volgende regel.

`char` variabelen kunnen via in- en uitvoer verwerkt worden.

```c
int main()
{
   char ch;

   printf("geef een teken\n");
   scanf("%c", &ch);
   printf("de code van %c is %d\n",ch,ch);

   return 0;
}
```

In dit voorbeeld wordt `%c` gebruikt bij in- en uitvoer. Het ingelezen teken wordt tweemaal op het scherm geplaatst: éénmaal als code en éénmaal als teken. Denk eraan dat we bij `scanf()` een `&` voor de variabele gebruiken en bij `printf()` niet. Indien het `&` teken[7] bij `scanf` vergeten wordt, dan geeft de compiler hiervoor geen foutmelding. Het programma zal dan wel starten maar de werking van de `scanf()` functie is onvoorspelbaar.

Het type `float`, `double` en `long double`

Voor reële getallen biedt de taal C de types:

parse_error	geheugenruimte:4 bytes bereik:3.4e-38 tot 3.4e+38
parse_error:	geheugenruimte:8 bytes bereik:1.7e-308 tot 1.7e+308
parse_error	geheugenruimte:12 bytes mantisse van 19 cijfers exponent: -16382 tot 16383 precisie: 80 bit

De aangegeven geheugenruimte en bereiken gelden voor de GNU C compiler op een 32 Linux platform.

Deze types kunnen zo in een declaratie gebruikt worden:

```c
float pi     = 3.14159;
double planck = 6.63e-34;
long double getal;
```

Wanneer een getalconstante met een decimale punt of een exponent genoteerd wordt, is dit een `double` constante. Wanneer het suffix `f` of `F` wordt toegevoegd is de constante van het type `float`. Indien het suffix `l` of `L` wordt bijgevoegd dan is de constante van het type `long double`.

```c
123.45F
.556L
46.
12e-3F
15.5E20
```

Deze constanten zijn allemaal van het reële type en kunnen aan een `long double`, `double` of een `float` variabele toegekend worden.

Er zijn 3 mogelijkheden om `double` en `float` variabelen op het scherm te plaatsen:

- `%f` gewone notatie
- `%e` exponent notatie
- `%g` gewone of exponent notatie

Als we `%g` gebruiken wordt indien mogelijk het getal in de gewone notatie op het scherm gedrukt; indien de exponent te groot of te klein verschijnt het getal in de exponent notatie op het scherm.

Voorbeeld:

[7] Het `&` teken is hier de neem het adres bewerking.

```
int main()
{
   float getal = 32000.0;

   printf("%f is gelijk aan %e\n",getal,getal);

   return 0;
}
```

Het opsommingstype

Het opsommingstype laat toe zelf symbolen als waarde te definiëren.

```
enum dagen =
{
   zondag,maandag,dinsdag,woensdag,donderdag,
   vrijdag,zaterdag
} vandaag, morgen;
```

De variabelen `vandaag` en `morgen` zijn van het type `enum dagen`. We kunnen hierin de namen van de dagen als waarde opslaan. Deze waarden worden als getallen opgeslagen. Het eerste symbool krijgt de waarde 0, het volgende de waarde 1 enzovoort. De uitdrukking `enum dagen` kan verder in het programma nog gebruikt worden voor de declaratie van andere variabelen.

```
enum dagen gisteren;
gisteren = woensdag;
```

We kunnen ook zelf een waarde koppelen aan elk symbool.

```
enum jaar
{
   Guldensporen=1302, Bastille=1789, VanGogh=1890
} feit;
```

De `sizeof()` functie

Tot slot nog een voorbeeld dat gebruik maakt van de ingebouwde functie `sizeof()`. Deze functie levert als resultaat de lengte (in bytes) van het type of de variabele die doorgegeven wordt. Het type van het resultaat is `int` (dit is afhankelijk van de implementatie).

```
#include <stdio.h>

int main()
{
   printf("lengte char: %d\n",sizeof(char));
   printf("lengte short int: %d\n",sizeof(short int));
   printf("lengte int: %d\n",sizeof(int));
   printf("lengte long int: %d\n",sizeof(long int));
   printf("lengte long long int: %d\n",sizeof(long long int));
   printf("lengte float: %d\n",sizeof(float));
   printf("lengte double: %d\n",sizeof(double));
   printf("lengte long double: %d",sizeof(long double));

   return 0;
}
```

De output van het bovenstaande programma is verkregen op een 32 bit Linux machine.

```
lengte char: 1
lengte short int: 2
lengte int: 4
lengte long int: 4
lengte long long int: 8
```

```
lengte float: 4
lengte double: 8
lengte long double: 12
```

Character strings, `#define`, `printf()`,`scanf()`

Met constanten bedoelen we het koppelen van een naam met een constante waarde. De `#define` opdracht wordt hiervoor gebruikt.

In dit hoofdstuk hebben we het verder over strings, `printf()` en `scanf()`.

Strings

Een string is een aaneenschakeling van tekens. Wanneer we een tekst op het scherm plaatsen, geven we een stringconstante door aan `printf()`. Met stringconstanten hebben we al kennis gemaakt.

```
printf("abcde");
```

De stringconstante `"abcde"` wordt gevormd door een tekst tussen dubbele aanhalingstekens. In feite worden de afzonderlijke tekens als een `char` constante opgeslagen.

```
'a' 'b' 'c' 'd' 'e' '\0'
```

Als laatste teken wordt nog de code 0 bijgevoegd. Dit geeft het einde van de string aan. Een stringconstante vraagt dus altijd 1 byte meer geheugen dan het aantal tekens. Deze 0 wordt gebruikt om het einde van de string te herkennen. Een functie die het aantal tekens in een string moet tellen, doorloopt de string en verhoogt een teller voor elk karakter in de string. De 0 code is dus het criterium om de herhaling te beëindigen.

Hier is een voorbeeld met strings en constanten.

```
/*
  De menselijke densiteit in kg/m3
 */
#define DENSITEIT 999

int main()
{
   float gewicht,volume;
   int grootte, letters;
   char naam[40];

   printf("geef je voornaam\n");
   scanf("%s", naam);
   printf("%s, geef je gewicht in kg\n",naam);
   scanf("%f",&gewicht);
   grootte = sizeof(naam);
   letters = strlen(naam);
   volume = gewicht/DENSITEIT;
   printf("%s, je volume is %10.4f m3\n",naam,volume);
   printf("je naam bestaat uit %d letters,\n",letters);
   printf("en we hebben %d bytes nodig",grootte);
   printf("om die op te slaan\n");

   return 0;
}
```

In dit programma treffen we een nieuw type variabele aan: array.

```
char naam[40];
```

Dit is de declaratie van de variabele `naam`. Hierdoor wordt er geheugenruimte gereserveerd voor 40 tekens. De `scanf()` functie zorgt ervoor dat deze variabele met een string gevuld wordt. Deze functie ontvangt 2

parameters: een stringconstante die een `%s` bevat en het adres van een string variabele. Deze `%s` aanduiding geeft aan dat een string van het toetsenbord gelezen wordt. Met het adres van de stringvariabele naam weet `scanf()` dat de ingegeven string terecht komt in de array naam. Wanneer een array variabele doorgegeven wordt aan `scanf()`, mag de adresoperator `&` niet gebruikt worden[8]. Al de tekens die op het toetsenbord ingegeven worden, komen in de stringvariabele terecht. Na het laatste teken plaatst `scanf()` nog de code `'\0'` om het einde van de string aan te duiden.

Dezelfde `%s` aanduiding wordt gebruikt om met `printf()` een string op het scherm te plaatsen.

We moeten toch wel opmerken dat er een verschil is tussen enkele en dubbele aanhalingstekens. De constante `"a"` is een char array die uit 2 tekens bestaat: `'a'` en `'\0'`, terwijl de constante `'a'` slechts een enkel teken is en bijgevolg van het type `char` is.

Met `sizeof(naam)` wordt er uitgerekend hoeveel bytes de variabele `naam` beslaat: dit zijn 40 bytes. Het programmavoorbeeld gebruikt een nieuwe functie: `strlen()`. Deze functie geeft ons de lengte van een string. De functie telt de tekens totdat de code 0 bereikt wordt. In het programma wordt `strlen()` gebruikt om na te gaan hoeveel tekens ingegeven zijn.

Tekstvervanging met `#define`

Bij de berekening van het volume zien we het symbool `DENSITEIT`. Dit is een constante die in de eerste programmaregel gedeclareerd wordt met `#define`. In C is het mogelijk om een tekst, die veel voorkomt, te koppelen aan een naam. Hiervoor dient de `#define` opdracht.

```
#define NAAM tekst
```

In de eerste fase van de vertaling worden alle namen die door `#define` zijn vastgelegd, vervangen door hun tekst. Deze taak wordt uitgevoerd door de preprocessor.

C programma —> preprocessor —> compiler

Hier zijn nog enkele voorbeelden:

```
#define PI 3.14159
#define DOLLAR '$'
```

De namen van deze constanten zijn in hoofdletters. Dit is niet verplicht, maar deze conventie wordt door veel C programmeurs gebruikt om het verschil tussen constanten en variabelen zichtbaar te maken.

We zien bij dit voorbeeld ook dat de `#define` werkt op basis van tekstvervanging. Telkens als de preprocessor een naam tegenkomt die met `#define` een betekenis heeft gekregen, wordt deze naam vervangen door de bijbehorende tekst. Men kan zelfs een hele opdracht bij een naam onderbrengen.

```
char slot[] = "tot ziens!";

int main()
{
   char naam[50];

   printf("geef je naam\n");
   scanf("%s", naam);
   printf("hallo %s\n", naam);
   printf("%d letters in %d bytes\n",
   strlen(naam),sizeof(naam) );
   printf("%d letters in %d bytes\n",
   strlen(slot),sizeof(slot) );
   printf("%s\n", slot);

   return 0;
}
```

Hiermee zien we dat de stringarray `slot` een extra byte nodig heeft voor de code 0.

[8] De naam van de array is al meteen het adres.

De conversietekens bij `printf()`

We bespreken hier de volledige mogelijkheden van de uitvoer met de `printf()` functie.

De volgende conversietekens kunnen in `printf()` gebruikt worden:

`%d %i` geheel decimaal getal

`%o` octaal

Het getal wordt niet vooraf gegaan door een 0.

`%x %X` hexadecimaal

Het getal wordt niet voorafgegaan door `0x` of `0X`. De waarden 10 tot 15 worden voorgesteld door abcdef of ABCDEF.

`%u` decimaal getal zonder teken

`%c` een enkel teken

`%s` een string

De tekens tot de code `'\0'` worden afgedrukt.

`%f` reëel zonder e notatie

Het formaat is `[-]m.dddddd`, het aantal cijfers na de decimale punt is 6.

`%e %E` reëel in e notatie

Het formaat is `[-]m.dddddde+xx` of `[-]m.ddddddE+XX`, het aantal cijfers na de decimale punt is 6.

`%g %G` reëel met of zonder e notatie

Het formaat is `%e` of `%E` als de exponent kleiner dan -4 of groter dan of gelijk aan de precisie is; gebruik anders `%f`. Nullen en/of een decimale punt worden niet afgedrukt.

`%p` pointer

Deze formaataanduiding wordt gebruikt om een pointer af te drukken. Het formaat is bepaald door de implementatie.

`%%` percent teken

Een extra controle is mogelijk met de volgende bijvoegsels. Deze bijvoegsels worden tussen de `%` en het conversieteken geplaatst.

`-` Het element wordt links in plaats van rechts gelijnd in het veld.

```
printf("%-10d", 123);
```

`+` Dit teken geeft aan dat voor het weer te geven getal altijd een plus- of een minteken moet worden gezet.

spatie Als het eerste teken geen plus- of minteken is, wordt een spatie voor het getal gezet.

`0` Deze nul zorgt ervoor dat het veld vooraan met nullen moet worden gevuld.

`#` alternatieve vorm

Met dit teken wordt een alternatieve vorm van uitvoer gespecificeerd. Als op `#` een `o` volgt, wordt ervoor gezorgd dat het eerste cijfer een 0 is. Volgt op `#` een `x` of een `X`, dan wordt voor een resultaat dat ongelijk is aan 0 als prefix `0x` of `0X` gezet. Is het teken `e`, `E`, `f`, `g` of `G`, dan heeft de uitvoer altijd een decimale punt. Is het teken `g` of `G`, dan worden de nullen achteraan niet verwijdert. Waarden die met deze vorm afgedrukt worden, kunnen altijd weer met een `scanf()` ingelezen. Dit is van belang voor de varianten van `printf()` en `scanf()` die van en naar bestanden lezen of schrijven.

getal minimum veldbreedte

Het geconverteerde argument wordt in een veld afgedrukt dat minimaal deze breedte heeft.

```
printf("%8d", 0x1234);
```

.getal precisie

Het getal is de precisie. Bij een string bepaalt dit getal het maximum aantal af te drukken tekens, bij reële getallen is de precisie het aantal af te drukken cijfers na de decimale punt.

```
printf("%6.2f",10/3);
```

`*` variabel formaat

In plaats van een getal voor de veldbreedte of de precisie mag ook het teken `*` gebruikt worden.

Dit betekent dat de veldbreedte en/of de precisie bepaald worden door variabelen. Deze variabelen moeten van het type `int` zijn.

```
printf("%*.*f", breedte,nauwk, 1/3 );
```

Eén van de volgende tekens mag vlak voor het conversieteken geplaatst worden:

`h short` in plaats van `int`

`l long` in plaats van `int`

```
printf("%ld", 0x1234L);
```

`L long double` in plaats van `double`

De conversietekens bij `scanf()`

Bij de `scanf()` functie wordt de invoer van het toetsenbord verwerkt. De conversie wordt bepaald door de conversietekens in de formaatstring. Elk van deze tekens neemt een deel van de ingave voor zich. De geconverteerde gegevens worden in variabelen geplaatst. Hiervoor worden na de formaatstring een reeks adressen van variabelen doorgegeven aan `scanf()`. Bij enkelvoudige variabelen is een `&` nodig om het adres van de variabele te berekenen.

In de formaatstring mogen buiten `%` tekens gevolgd door een conversieteken ook andere tekens staan:

- Spaties of tabs: deze worden genegeerd.
- Gewone tekens (geen `%`); deze moeten overeenstemmen met het volgende niet-witruimteteken van de invoer.

Een conversiespecificatie (`%` teken met een conversieteken) regelt de conversie van het eerstvolgende invoerveld. Als tussen `%` en het conversieteken een `*` wordt geplaatst, zoals in `%*s`, dan wordt de toekenning onderdrukt: het invoerveld wordt dan eenvoudig overgeslagen en er vindt geen toekenning plaats.

Een invoerveld wordt gedefiniëerd als een string van niet-witruimtetekens. Zo 'n veld strekt zich uit tot aan het eerstvolgende witruimteteken of eindigt, als een veldbreedte is opgegeven, op de plaats waar die veldbreedte is bereikt. Dit betekent dus dat `scanf()` om zijn invoer te vinden over de regelgrenzen leest, omdat een newline als witruimte geldt. De witruimtetekens zijn: spatie, tab, newline, carriage return en formfeed.

Voor de conversietekens `d`, `i`, `n`, `o`, `u` en `x` mag een h of een `l` worden gezet: met `h` wordt aangegeven dat een `short` variabele gevuld moet worden; met `l` wordt aangegeven dat een `long` moet gevuld worden. Op dezelfde manier mag voor de conversietekens `e`, `f` en `g` een `l` of een `L` geplaatst worden: `l` leest een `double` en `L` leest een `long double`.

De volgende conversietekens kunnen bij `scanf()` gebruikt worden:

`%d` geheel decimaal getal

`%i` geheel getal

Het getal mag in decimaal, octaal of hexadecimaal ingegeven worden. Een decimaal getal start niet met een 0. Een octaal getal start met een 0 en een hexadecimaal getal start met `0x` of `0x`.

`%o` octaal geheel getal

Het getal is al dan niet voorafgegaan door een 0.

%x %X hexadecimaal geheel getal

Het getal is al dan niet voorafgegaan door `0x` of `0X`.

%u decimaal getal zonder teken

%c tekens

De volgende invoertekens worden in de opgegeven array gezet, en wel tot aan het in het breedteveld aangegeven aantal. Dit aantal is bij verstek 1. Er wordt geen `'\0'` toegevoegd bij de ingelezen tekens. Het gebruikelijke overslaan van witruimtetekens wordt onderdrukt. Om het volgende niet-witruimteteken te lezen moet `%1s` gebruikt worden.

%s een string van niet-witruimtetekens

Aan de ingelezen tekens wordt nog de code `'\0'` bijgevoegd. De array variabele moet groot genoeg zijn om al de ingegeven tekens op te slaan. Als er meer ingegeven wordt dan er plaats is in de array, gebeuren er rare dingen.

%f %e % geen reëel getal

Het invoerformaat is: een optioneel teken, een string van cijfers, mogelijk met een decimale punt en een optioneel exponentveld met een `E` of een `e`, gevolgd door een integer, mogelijk met teken.

%p pointer

Een pointer wordt ingelezen. Het formaat is zoals het formaat bij het afdrukken met `printf()`. Dit wordt bepaald door de implementatie.

%n aantal invoervelden

In een meegeleverde `int` variabele wordt het aantal tot nu toe door deze `scanf()` ingelezen velden geplaatst. Er wordt geen invoer gelezen en de teller die intern in de `scanf()` functie het aantal gelezen velden telt, wordt niet verhoogd.

`[...]` Dit correspondeert met de langste niet-lege string van invoertekens uit de verzameling tussen de haken. Aan het einde wordt een `'\0'` toegevoegd. Met `[]...]` wordt het teken `]` in de verzameling opgenomen.

`[^...]` Dit correspondeert met de langste niet-lege string van invoertekens die niet in de verzameling tussen de haken voorkomen. Aan het einde wordt een `'\0'` toegevoegd. Met `[^]...]` wordt het teken `]` in de verzameling opgenomen.

%% percent teken

Er vindt geen toekenning plaats.

De toekenning, operatoren en uitdrukkingen

De toekenning is een essentieel element bij de imperatieve talen. Met deze opdracht kunnen we een waarde opslaan in een variabele. Zolang er geen nieuwe toekenning plaats heeft voor deze variabele, behoudt de variabele zijn waarde. De waarde die aan een variabele wordt toegekend, kan niet alleen een constante zijn maar ook de inhoud van een andere variabele of het resultaat van een uitdrukking. Een uitdrukking bestaat uit een aantal constante waarden en inhouden van variabele die met elkaar worden gecombineerd door operatoren.

Het volgende programma toont hoe we met behulp van operatoren en uitdrukkingen een waarde kunnen toekennen aan een variabele.

```
int main()
{
   float celsius,fahrenheit;

   printf("Temperatuurtabel\n");
   celsius = 0;
   while(celsius <= 100)
   {
```

```
        fahrenheit = 9.0/5*celsius + 32;
        printf("%4.1f celsius is %4.1f fahrenheit\n",
        celsius, fahrenheit);
        celsius = celsius + 5;
    }
    return 0;
}
```

Dit programma plaatst een omzettingstabel van graden Celsius naar graden Fahrenheit op het scherm. Alle Celsius waarden van 0 tot 100 worden in een stap van 5 omgezet naar Fahrenheit. De herhaling van deze berekening wordt met een `while` opdracht uitgevoerd. De herhaling gaat verder zolang de voorwaarde die bij de `while` vermeld wordt, waar is. De opdrachten die in deze herhaling betrokken zijn, worden tussen accolades vermeld na de `while`. De eerste Celsius waarde die op het scherm verschijnt, is de waarde 0. De laatste is 100. Hierna wordt de variabele `celsius` nog een keer verhoogd tot 105. De voorwaarde die bij de `while` hoort, is dan niet meer waar en de herhaling stopt.

We bespreken de volgende punten:

Toekenning

De algemene vorm van een toekenning is:

variabelenaam = uitdrukking;

De waarde van de uitdrukking wordt uitgerekend en dan in de variabele geplaatst. Er kan ook een waarde aan meerdere variabelen toegekend worden.

```
a = b = c = 1;
```

Bij deze toekenning wordt eerst `c` 1, daarna `b` en dan pas `a`.

Rekenkundige operatoren

Rekenkundige bewerkingen kunnen in uitdrukkingen toegepast worden. Als basisoperatoren hebben we `+`, `-`, `*` en `/`. In het voorbeeld

```
(-b + c)/a
```

is `-` een unaire operator (werkt op 1 operand), `+` en `/` zijn binaire operatoren (werkt op 2 operanden). Bij de deling wordt dezelfde operator voor gehele en reële getallen gebruikt.

`39/5` wordt `7`

`39./5` wordt `7.8`

Als de deler of het deeltal reëel is, is het quotiënt ook reëel. Om de rest van een gehele deling te berekenen, wordt de `%` operator[9] toegepast.

`39%5` wordt `4`

De rekenkundige operatoren kunnen in twee soorten verdeeld worden:

binair: `+ - * /` unair: `+ -`

In de uitdrukking `a + +(b - c)` zorgt de unaire `+` ervoor dat eerst het verschil van `b` en `c` uitgerekend wordt. Hierdoor kan een overflow van `a+b` vermeden worden.

Er bestaat geen operator voor machtsverheffing.

Dit zijn de prioriteiten met telkens de volgorde van uitvoering als meerdere operatoren van dezelfde prioriteit voorkomen.

prioriteit	operator	
hoog	parse_error	van links naar rechts
	`-` `+` unair	

[9] Dit is typisch C: meerdere betekenissen voor hetzelfde letterteken.

	* / %	
	+ - binair	van links naar rechts
laag	parse_error	van rechts naar links

Met 1 verhogen of verlagen (++ en --)

C kent een speciale notatie om een variabele met 1 te verhogen of te verlagen. Dit kan handig zijn zeker als de uitdrukking die de variabele voorstelt lang is.

Hier zijn enkele voorbeelden:

```
a++;
a--;
```

ofwel

```
++a;
--a;
```

De operatoren ++ en -- doen hetzelfde als:

```
a = a + 1;
a = a - 1;
```

Een ++ of -- is soms moeilijk te interpreteren. Wat betekent de volgende uitdrukking?

```
x*y++
```

Dit is hetzelfde als x*(y++) en dus niet (x*y)++. We zonderen y++ af met haken omdat ++ een hogere prioriteit heeft dan *. De tweede notatie is trouwens zinloos omdat we alleen een variabele kunnen verhogen en geen uitdrukking.

De ++ en -- operatoren kunnen voor of na de variabele geplaatst worden. Dit betekent vooraf of achteraf verhogen.

- postfix notatie

```
i = 0; j = i++;/* j wordt 0 */
```

eerst waarde gebruiken en daarna verhogen

- prefix notatie

```
m = 0; n = ++m;/* n wordt 1 */
```

eerst verhogen en daarna waarde gebruiken

In het volgende voorbeeld wordt de verhoging van i in de voorwaarde ingebouwd.

```
int main()
{
   int i = 0;

   while(++i < 20)
   {
      printf("%d\n", i );
   }

   return 0;
}
```

Deze notatie levert kortere programma's op. Het nadeel is dat deze programma's minder goed leesbaar zijn en dat er soms ongewenste zijeffecten worden gecreëerd.

Bitoperatoren

Weinig programmeertalen hebben operatoren voor bewerkingen op bitniveau. De taal C vormt hierop een uitzondering. Dit is begrijpelijk als men weet dat de ontwerpers van C een taal hebben ontworpen om assembler te vervangen.

Het ontstaan van de bitoperatoren in C is te verklaren door het feit dat de hoeveelheid geheugen in de eerste generatie computers heel beperkt was. Zo werden bits gebruikt om de ja/neen waarden op te slaan. Tegenwoordig worden de bitoperatoren hoofdzakelijk gebruikt voor IO bewerkingen in microcontrollers.

Deze bitoperatoren mogen uitsluitend op gehele getallen toegepast worden. Dit zijn de types `char`, `short`, `int` en `long` met of zonder teken.

parse_error	bitsgewijs en
parse_error	bitsgewijs inclusieve of
parse_error	bitsgewijs exclusieve of
parse_error	verschuif naar links
parse_error	verschuif naar rechts
parse_error	één complement (unair)

Alleen de laatste operator is unair, de overige zijn binair; ze vragen twee operands.

Om bit 3 in een variabele op 1 te zetten schrijven we:

```
x = x | 010;
```

Om dezelfde bit terug op 0 te zetten schrijven we:

```
x = x & 0177767;
```

ofwel

```
x = x & ~010;
```

De eerste vorm kan enkel gebruikt worden voor een variabele van het type `int`. De tweede vorm kan voor elk geheel type gebruikt worden[10].

De bitoperator ^ levert in een bit het resultaat 1 als de twee bits uit de operands verschillend zijn.

De verschuifoperatoren << en >> zorgen ervoor dat de bits van de linker operand verschoven worden. Het aantal bits dat verschoven wordt, is afhankelijk van de rechter operand. Bij `x << 3` wordt de waarde 3 plaatsen naar links verschoven. De vrijgekomen bits worden met 0 bits opgevuld. Het resultaat is in dit geval hetzelfde als vermenigvuldigen met 8.

Bij het verschuiven naar rechts is er een verschil tussen tekenloze en getallen met teken. Bij `unsigned` waarden worden aan de linkerkant nullen ingeschoven. Dit noemt men logisch verschuiven. Bij waarden met teken wordt links de tekenbit ingeschoven (rekenkundig verschuiven) of bij sommige implementaties wordt in dit geval ook een nul ingeschoven.

De operator ~ zorgt voor de omkering van alle bits van het getal. Een 0 wordt 1 en een 1 wordt 0.

Samentrekking van toekenning en operator

Dit zijn kortere vormen voor toekenningen.

parse_error	wordt	parse_error
parse_error		parse_error
parse_error		parse_error
parse_error		parse_error
parse_error		parse_error
parse_error		parse_error
parse_error		parse_error

[10] Alle voorbeelden hier zijn in het octaal.

parse_error	parse_error
parse_error	parse_error
parse_error	parse_error

Ook deze notaties leveren kortere programma's op.

Let wel op de prioriteiten:

```
a *= b + 2;
```

betekent

```
a = a * (b + 2);
```

en niet

```
a = a * b + 2;
```

Uitdrukkingen

Dit is een combinatie van bewerkingen, constanten en variabelen. Een uitdrukking stelt steeds een waarde voor. Deze waarde kan berekend worden door de bewerkingen volgens hun prioriteiten uit te rekenen. Enkele voorbeelden:

```
5
-125
1 + 1
a = 3
b = ++b % 4
c > 3.14
```

Ook de toekenning stelt een waarde voor. Dit is de waarde die aan de variabele toegekend wordt.

Opdrachten

Opdrachten zijn de bouwstenen van een programma. Elke opdracht voert een actie uit.

```
/* de som van de eerste 20 getallen */
int main()
{
   int teller, som;/* declaratie*/

   teller = 0;      /* toekenning*/
   som = 0;         /*   idem*/
   while (teller++ < 20)      /* while*/
   {
      som = som + teller;     /*   opdracht*/
   }
   printf("som = %d\n",som); /* functie oproep*/

   return 0;
}
```

Elke opdracht wordt met een `;` afgesloten.

Een samengestelde opdracht bestaat uit meerdere opdrachten tussen `{` en `}`.

```
while (i++ < 100)
   j = i * i; // alleen deze opdracht in herhaling
printf("%d\n",j);
```

In het vorige voorbeeld hoort er bij de `while` slechts een opdracht. In het volgend voorbeeld plaatsen we twee opdrachten bij de `while`.

```
while (i++ < 100)
{
   j = i * i;
   printf("%d\n",j);
}
```

Probeer altijd de accoladen `{` en `}` te gebruiken bij de controlestructuren. Dit levert beter leesbare programma's op.

Typeomzetting

Automatische omzetting

Bij het uitrekenen van uitdrukkingen waarin constanten en variabelen van hetzelfde type voorkomen, is geen typeomzetting nodig. Wanneer er verschillende types voorkomen, gebeurt er automatisch een omzetting van een lager type naar een hoger type.

De omzetting vindt alleen maar plaats als er geen verlies van informatie is. Bij de uitdrukking `f + i` wordt de `int` variabele automatisch omgezet tot `float` omdat `f` van het type `float` is. Bij de toekenning gebeurt er een omzetting naar het type van de variabele, die de waarde ontvangt. Dit betekent dus een promotie of degradering. Dit laatste kan problemen geven, wanneer de waarde niet in het bereik past. In dit geval kan de compiler een waarschuwing geven.

```
char k;

k = 200 + 321;
k = 2.3e45;
```

cast bewerking

Dit is een geforceerde omzetting.

```
int m;

m = 1.6 + 1.5;/* geeft 3 */
m = (int) 1.6 + (int) 1.5;/* geeft 2 */
```

Het type wordt tussen haken voor de om te zetten waarde geplaatst. De omzetting `(int)` geeft afkapping en geen afronding.

Keuze maken

De `if` opdracht

Met de `if` opdracht kunnen we de uitvoering van het programma beïnvloeden. Afhankelijk van een voorwaarde wordt de ene of de andere opdracht uitgevoerd.

In het volgende voorbeeld wordt de `if` gebruikt om na te gaan of een getal oneven is.

```
void main()
{
   int teller = 0;
   int som    = 0;

   while (teller++ < 100)
   {
      if ( teller % 2 != 0)
      {
         som += teller;
      }
   }
   printf("de som van de oneven getallen is %d\n",som);
}
```

De algemene vorm is:

```
if (uitdrukking)
   opdracht
```

Het resultaat van de uitdrukking bepaalt of de opdracht al dan niet uitgevoerd wordt.

niet 0 : uitvoeren

0 : niet uitvoeren

Het is mogelijk om meerdere opdrachten bij een `if` te plaatsen. We plaatsen de opdrachten tussen accolades.

```
if (a == b)
{
   printf("twee gelijke getallen:\n");
   printf("%d en %d\n", a, b);
}
```

We kunnen ook een opdracht laten uitvoeren als de voorwaarde niet waar is. Dit wordt aangegeven door het woord `else`.

```
if (a == 0)
   printf("het getal is nul\n");
else
   printf("het getal is niet nul\n");
```

De algemene vorm is:

```
if (uitdrukking)
   opdracht
else
   opdracht
```

Als opdracht bij een `if` of `else` kan een andere `if` gebruikt worden.

```
if (a == 0)
{
   printf("het getal is nul\n");
}
else
{
   if (a > 0)
   {
      printf("het getal is positief\n");
   }
   else
   {
      printf("het getal is negatief\n");
   }
}
```

Indien we veel `if` opdrachten met elkaar combineren, kunnen we de insprong beter weglaten.

```
if (bedrag < 1000)
   korting = 0;
else if (bedrag < 2500)
   korting = 2;
else if (bedrag < 5000)
   korting = 5;
else if (bedrag < 10000)
   korting = 8;
else
```

```
   korting = 10;
bedrag *= 1 - korting/100;
```

De structuur in het vorige voorbeelden komt in praktijk veel voor. In deze structuur wordt één opdracht uit vele uitgevoerd.

Wanneer een `else` volgt na meerdere `if` opdrachten, kunnen we ons afvragen bij welke `if` deze `else` hoort.

```
if (getal > 5)
if (getal < 10)
   printf("goed\n");
else
   printf("slecht\n");
```

Bij dit programma zouden we kunnen denken dat de `else` bij de eerste `if` hoort, maar deze interpretatie is fout. Een `else` hoort steeds bij de laatste `else`-loze `if`.

Dit is de verbeterde versie:

```
if (getal > 5)
   if (getal < 10)
      printf("goed\n");
   else
      printf("slecht\n");
```

Als we de `else` toch bij de eerste `if` willen plaatsen, dan kan dit zo:

```
if (getal > 5)
{
   if (getal < 10)
      printf("goed\n");
}
else
   printf("slecht\n");
```

Opnieuw moeten we stellen dat het beter is om de accolades altijd te schrijven. Het laatste voorbeeld wordt dan uiteindelijk:

```
if (getal > 5)
{
   if (getal < 10)
   {
      printf("goed\n");
   }
}
else
{
   printf("slecht\n");
}
```

Relationele operatoren

Met deze operatoren kunnen we uitdrukkingen schrijven die vergelijkingen uitvoeren.

```
<  kleiner dan
>  groter dan
<= kleiner dan of gelijk aan
>= groter dan of gelijk aan
== gelijk aan
!= verschillend van
```

Alleen waarden van de types `(un)signed char`, `short`, `int`, `long`, `pointer`, `float` en `double` kunnen met elkaar vergeleken worden.

Het resultaat van deze vergelijkingen is 1 (waar) of 0 (niet waar); het resultaat is van het type `int`. C kent dus geen boolse constanten of variabelen. We kunnen dit uitproberen met de volgende opdracht.

```
printf("waar %d, niet waar %d\n", 5>1, 0!=0);
```

Dit voorbeeld toont dat we gehele getallen krijgen als resultaat van vergelijkingen.

Let wel op voor het verschil tussen `=` (toekenning) en `==` (test gelijkheid). Het verwisselen van deze twee operatoren is een veel voorkomende fout, die niet door alle compilers gesignaleerd worden.

`a = 5` levert `5`

`a == 5` levert `1` als `a` gelijk aan `5` anders `0`

Deze twee operatoren worden verschillend geschreven omdat ze tegelijkertijd bij een `if` gebruikt kunnen worden.

```
if ((a = b) == 0)
```

Deze opdracht plaatst eerst de inhoud van `b` in `a` en test dan of deze waarde gelijk is aan 0. `a = b` staat tussen haken omdat de toekenning een lagere prioriteit heeft dan de gelijkheidsvergelijking. Soms wordt de vergelijking verschillend van 0 weggelaten.

`if (aanwezigen != 0)` is identiek aan `if (aanwezigen).`

De laatste notatie die wel korter is, is niet aan te bevelen wegens de slechte leesbaarheid.

De prioriteit van relationele operatoren is lager dan die van rekenkundige operatoren. De uitdrukking `a + b == 0` kunnen we dus als `(a + b) == 0` interpreteren.

Logische operatoren

Met deze operatoren kunnen we meerdere voorwaarden logisch met elkaar koppelen.

```
// tel kleine letters in een regel
int main()
{
   char t;
   int aantal = 0;

   while ( ( t=getchar() ) != '\n')
   {
      if (t >= 'a' && t <= 'z')
         aantal++;
   }
   printf("het aantal is %d\n", aantal);

   return 0;
}
```

De uitdrukking bij de `while` kent eerst een waarde toe aan de variabele `t`. Deze waarde komt van de functie `getchar()`. Deze functie wacht tot een toets ingedrukt wordt en levert de code van deze toets als resultaat. Hierna wordt er getest of de ingegeven toets geen return is. Deze `while` herhaling gaat verder totdat alle tekens van de ingaveregel verwerkt zijn. Dit programma telt het aantal kleine letters in een regel tekst.

Bij de `if` opdracht zien we dat de twee voorwaarden gekoppeld zijn met de `&&` operator. De opdracht bij de `if` wordt dus enkel uitgevoerd als de twee voorwaarden waar zijn.

Er zijn drie logische operatoren:

- `&&` logische en
- `||` logische of
- `!` logische niet

De werking is:

`uitdr1 && uitdr2`

waar als beide uitdr1 en uitdr2 waar zijn

`uitdr1 || uitdr2`

waar als ofwel een van de twee ofwel beide uitdrukkingen waar zijn

`! uitdr1`

waar als uitdr1 niet waar is

We mogen de logische operatoren niet verwarren met de bitoperatoren `&`, `|` en `~`. De bitoperatoren werken op de bits apart en de logische operatoren worden uitgevoerd op de getalwaarden. Bij deze laatsten is het alleen van belang of getal nul is of niet.

```
4 && 2  // geeft 1
4 & 2  // geeft 0
4 || 2 // geeft 1
4 | 2 // geeft 6
```

Hier zijn nog enkele voorbeelden:

```
6 > 1 && 10 == 5    // niet waar
6 > 1 || 10 == 5   //waar
!(3 > 9)  // waar
of 3 <= 9
```

De volgorde van evaluatie is steeds van links naar rechts. Als het eindresultaat al vastligt na evaluatie van de eerste uitdrukking, wordt de tweede niet meer geëvalueerd.

```
0 && uitdr2  // geeft altijd 0
1 || uitdr2  // geeft altijd 1
```

Deze kortsluitmogelijkheid is handig om bepaalde fouten te vermijden.

```
if ( n != 0 && 12/n == 2)
{
   printf("n is 5 of 6\n");
}
```

Hier wordt de deling door `n` enkel uitgevoerd als `n` verschillend is van 0.

De prioriteiten van logische operatoren zijn:

- `!` heeft een hogere prioriteit dan && en `||`
- `&&` heeft een hogere prioriteit dan `||` .

De logische operatoren hebben een lagere prioriteit dan relationele operatoren.

Dus `i == 1 && j == 2    || i == j` is hetzelfde als `((i == 1) && (j == 2)) || (i == j)`

Let erop dat de bitoperatoren een lagere prioriteit hebben dan de relationele operatoren. Hierdoor moeten we in de volgende voorwaarde haken gebruiken.

```
(x & 0x8) == 0
```

Deze voorwaarde test of bit 3 nul is.

Conditionele uitdrukking `?:`

Deze opdracht maakt een keuze uit twee waarden afhankelijk van een voorwaarde.

```
a = (b < 0) ? -b : b;
```

We kunnen dit ook met een `if` schrijven.

```
if (b < 0)
{
   a = -b;
```

```
}
else
{
   a = b;
}
```

De conditionele uitdrukking bestaat uit:

uitdr1 ? uitdr2 : uitdr3

Het resultaat van deze uitdrukking is uitdr2 als uitdr1 waar is, anders is het resultaat uitdr3.

Tenslotte nog een voorbeeld waarbij twee getallen in stijgende volgorde op het scherm geplaatst worden.

```
printf("%d,%d\n", (a > b) ? b : a, (a > b) ? a : b );
```

Meerdere keuzemogelijkheden: `switch`

Wanneer we een keuze uit meerdere mogelijkheden maken, dan is de `switch` opdracht de beste oplossing.

```
int main()
{
   char letter;

   printf("geef een letter en ik geef je een vogelnaam\n");
   while ( ( letter=getchar() ) != '#')
   {
      switch (letter)
      {
         case 'a' :
            printf("aalscholver, phalacrocorax carbo\n");
            break;
         case 'b' :
            printf("bontbekplevier, charadrius hiaticula\n");
            break;
         case 'c' :
            printf("citroensijs, serinus citrinella\n");
            break;
         case 'd' :
            printf("duinpieper, anthus campestris\n");
            break;
         case 'e' :
            printf("eidereend, somateria mollissima\n");
            break;
         default :
            printf("vandaag alleen van a to e\n");
            break;
      }
   }

   return 0;
}
```

Dit programma leest een letter in en voert dan een actie uit die bij deze letter hoort. Dit wordt herhaald tot een `#` ingegeven wordt. De `switch` opdracht neemt de inhoud van de variabele letter en zoekt dan met welke constante deze waarde overeenkomt. De constanten worden elk na `case` vermeld. De opdracht die bij de gevonden constante hoort, wordt uitgevoerd. Indien de inhoud van letter niet als constante voorkomt, dan wordt de `default` opdracht uitgevoerd. In dit programma wordt er dus voor elke ingegeven letter een `printf()` opdracht uitgevoerd.

Dit is de algemene vorm:

```
switch ( uitdrukking )
{
   case constante1 :
      opdrachten;
      break;
   case constante2 :
      opdrachten;
      break;
   default :
      opdrachten;
   break;
}
```

De uitdrukking en constanten moeten van type `int` of `char` zijn. We kunnen hier dus geen `float` of `double` gebruiken. De opdrachten `break` en `default` mogen weggelaten worden. Bijvoorbeeld het uitvoeren van dezelfde opdracht voor 2 constanten:

```
case 'F' :
case 'f' :
   printf("fitis, phylloscopus trochilus\n");
   break;
```

Als `default` met bijbehorende opdracht en `break` weggelaten worden, dan wordt geen opdracht uitgevoerd wanneer de geteste waarde niet als constante voorkomt.

Lussen en andere controlemiddelen

`while` herhalingsopdracht

Met deze herhalingsopdracht hebben we al kennis gemaakt. De algemene vorm is:

```
while (uitdrukking)
   opdracht;

while (uitdrukking)
{
   opdracht1;
   opdracht2;
}
```

De opdrachten worden herhaald zolang de voorwaarde waar is. In de opdrachten moet er steeds één voorkomen die de waarde van de geteste uitdrukking verandert. Indien dit niet zo is, stopt de herhaling nooit.

In de volgende voorbeelden wordt `i` als lusteller gebruikt. In elk voorbeeld wordt `i` op een andere wijze verhoogd.

- geen einde:

  ```
  i = 1;
  while (i < 10)
  {
     printf("dit is i: %d\n", i);
  }
  ```

- resultaat: 2 - 9

  ```
  i = 1;
  while (++i < 10)
  {
     printf("dit is i: %d\n", i);
  }
  ```

- resultaat: 2 - 10

```
i = 1;
while (i++ < 10)
{
   printf("dit is i: %d\n", i);
}
```

- resultaat: 1 - 9

```
i = 1;
while (i < 10)
{
   printf("dit is i: %d\n", i);
   i++;
}
```

De structuur van het laatste voorbeeld:

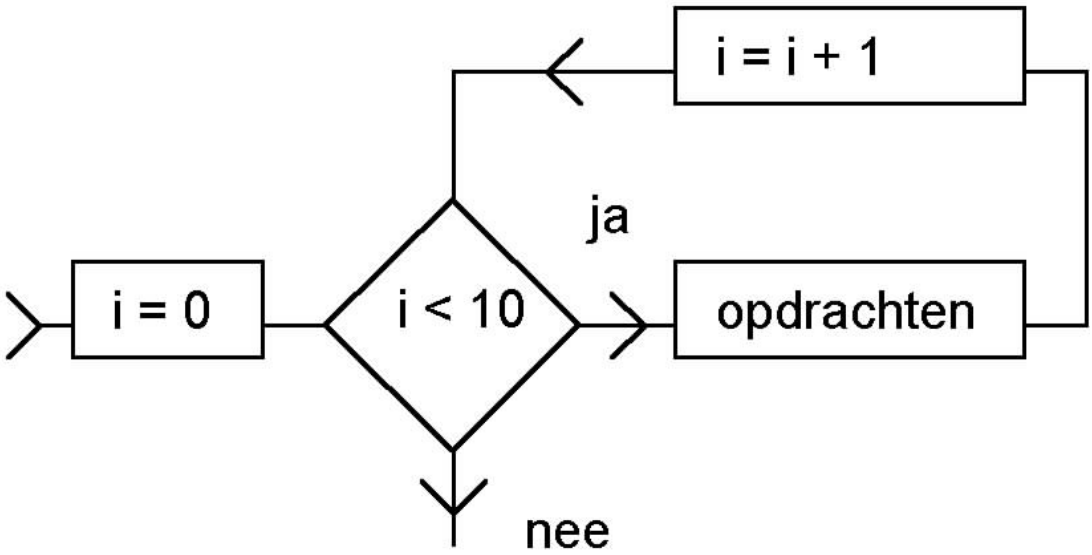

Figure 1: Werking `for` in flow-chart

Deze herhaling bestaat uit de initialisatie van de lusteller, het testen van de eindvoorwaarde en het verhogen van de lusteller.

`for` herhalingsopdracht

Het laatste voorbeeld van `while` is nu met een `for` herschreven zonder dat de werking verandert. Ook voor dit voorbeeld[11] geldt het stroomdiagramma.

```
for (int i = 1; i < 10; i++)
{
   printf("dit is i: %d\n", i);
}
```

Dit zijn nog andere voorbeelden:

[11] Zoals je ziet, is de declaratie van de lusteller binnen de `for` geplaatst. Dit mag sinds de C99 standaard. Hierdoor is de schrijfwijze van de `for` hetzelfde geworden zoals die in C++ en Java.

- een lege opdracht in herhaling

```
for (int n = 1; n <= 10000; n++)
{

}
```

- stap verschillend van 1

```
for (int n = 2; n < 100; n += 11)
{
   printf("%d\n", n);
}
```

- stap verhogen met `*`

```
for (float bedrag = 100; bedrag < 200; bedrag *= 1.08)
{
   printf("bedrag: %.2f\n", bedrag);
}
```

- `char` als lusteller

```
for (char t = 'a'; t <= 'z'; t++)
{
   printf("%c", t);
}
```

- een opdracht minder in `for`

```
for (int u = 1; u < 1000; )
{
   u *= 2;
}
```

- geen opdrachten in `for`

```
for ( ; ; )
{
   printf("hallo\n");
}
```

De algemene vorm van de `for` opdracht is:

```
for ( initialisatie ; test ; aanpassen )
   opdracht
```

Tussen de haakjes van de `for` opdracht kunnen we 3 opdrachten onderbrengen. Als we bijvoorbeeld een extra opdracht willen laten uitvoeren bij de initialisatie, dan wordt deze opdracht met een komma bijgevoegd. Dit noemt men in C de komma bewerking.

De 3 puntkomma's moeten altijd geschreven worden.

```
for (int j=1, float bedrag = 100; bedrag < 200; j++, bedrag *= 1.08)
{
   printf("jaar: %d bedrag: %.2f\n", j, bedrag);
}
```

`do while` herhalingsopdracht

Bij deze herhalingsopdracht wordt de voorwaarde getest nadat de opdracht uitgevoerd is. Dit betekent dat de opdracht minstens éénmaal uitgevoerd wordt, ook als de voorwaarde steeds false is.

De algemene vorm is:

```
do
   opdracht
while ( voorwaarde );
```

In het volgende voorbeeld worden de tekens van een ingegeven regel omgezet in de decimale ASCII code.

```
do
{
   scanf("%c", &teken);
   printf("%c heeft als code %d\n", teken, teken);
} while (teken != '\n');
```

break en continue bij herhalingsopdrachten

Bij complexere problemen is het wenselijk om een herhaling voortijdig af te breken of te herstarten. Hiervoor voorziet C de opdrachten `break` en `continue`. Met een `break` kunnen we op een handige manier de herhaling stopzetten midden in een reeks opdrachten. Dit probleem kan ook opgelost worden zonder gebruik te maken van `break`. Dit vraagt dan wel iets meer denkwerk. Het is dan ook om deze reden dat niet alle programmeertalen deze mogelijkheid kennen.

break

In het volgende voorbeeld wordt in de herhaling telkens een getal ingelezen en het kwadraat hiervan op het scherm gedrukt. De herhaling gaat verder totdat ofwel het ingegeven getal nul is ofwel het aantal ingelezen getallen groter dan 20 is. Hier heeft de `while` opdracht een voorwaarde die steeds waar is. Het stopzetten van de herhaling wordt met `break` uitgevoerd.

```
i = 0;
while (1 == 1)
{
   printf("geef een getal: ");
   scanf("%d", &getal);
   if (getal == 0)
   {
      break;
   }
   printf("kwadraat van %d is %d\n",
      getal, getal*getal);
   if (++i > 20)
   {
      break;
   }
}
```

Het is mogelijk om dit te herschrijven zonder de `break` opdracht.

```
#define FALSE 0
#define TRUE 1

einde = FALSE;
i = 0;
while ( !einde )
{
   printf("geef een getal: ");
   scanf("%d", &getal);
   if (getal == 0)
   {
      einde = TRUE;
   }
   else
   {
```

```
            printf("kwadraat van %d is %d\n",
               getal, getal*getal);
            if (++i > 20)
            {
               einde = TRUE;
            }
         }
      }
```

continue

Dit is een opdracht die de uitvoering van de herhalingsopdracht laat herstarten. Anders geformuleerd: de opdrachten na `continue` worden overgeslagen. In het volgende voorbeeld wordt in de `while` opdracht de verwerking van spaties overgeslagen.

```
while( (ch = getchar() ) != EOF)
{
   if (ch == ' ')
   {
      continue;
   }
   putchar( ch );
   teller++;
}
```

Dit voorbeeld kan herschreven worden zonder `continue`.

```
while( (ch = getchar() ) != EOF)
{
   if (ch != ' ')
   {
      putchar( ch );
      teller++;
   }
}
```

goto

De `goto` opdracht maakt het mogelijk om naar een andere plaats in het programma te springen. Dit is een opdracht die nog stamt uit het FORTRAN tijdperk. Deze opdracht wordt bijna nooit meer gebruikt. De `goto` is inmiddels overbodig geworden. Met gestructureerd programmeren kan men immers iedere mogelijke constructie opbouwen zonder `goto` te gebruiken. Programma's met `goto` zijn dikwijls moeilijk leesbaar en daardoor slecht onderhoudbaar. Gebruik daarom geen `goto` en beschouw het als onbestaande. Om deze redenen wordt de `goto` niet verder besproken [12].

Functies

Kennismaking

Bij één van de eerste programmavoorbeelden hebben we al kennisgemaakt met functies. Een functie groepeert meerdere opdrachten bij een naam. Deze opdrachten kunnen uitgevoerd worden, als we de functienaam als een gewone opdracht gebruiken.

In het volgende voorbeeld wordt de functie `lijn()` gebruikt om tweemaal een lijn van sterretjes op het scherm te schrijven.

```
void lijn()
{
   for (int i=0; i<18; i++)
```

[12] Edsger W. Dijkstra was indertijd een hevig tegenstander van het gebruik van de `goto`.

```
    {
        printf("*");
    }
    printf("\n");
}

int main()
{
    lijn();
    printf("Dit is de cursus C\n");
    lijn();

    return 0;
}
```

Uit dit voorbeeld blijkt ook dat we variabelen kunnen declareren binnen de functie. De variabele `i` mag alleen maar gebruikt worden binnen de functie. De declaratie van de variabelen binnen de functie worden vlak na de openingsaccolade vermeld. Deze vorm van lokale variabelen is trouwens niet beperkt tot functies. De syntax is algemeen geldig: na elke openingsaccolade die opdrachten groepeert, mogen we variabelen declareren[13].

Parameters

We kunnen de flexibiliteit van een functie verhogen als we bij de oproep een waarde doorgeven. Dit betekent dat we een gedeelte van de werking verschuiven naar de oproep van de functie. In het voorbeeld wordt vastgelegd dat de functie `spatie()` een waarde van het type `int` ontvangt bij de oproep. De waarde komt terecht in de variabele `aantal` en wordt door de functie gebruikt om het aantal spaties te bepalen. Bij de oproep wordt de door te geven waarde tussen de functiehaken geplaatst.

```
void spatie(int aantal)
{
    for (int i=0; i < aantal; i++)
    {
        printf(" ");
    }
}

int main()
{
    printf("Dit is de cursus C\n");
    spatie(16);
    printf("+++\n");

    return 0;
}
```

In verband met parameters kent men de volgende terminologie:

formele parameter de variabele die de doorgegeven waarde ontvangt

actuele parameter de werkelijke waarde die doorgegeven wordt.

We kunnen een functie met meerdere parameters voorzien. De formele en actuele parameters worden gescheiden door komma's.

```
tlijn(char t, int n)
{
    for (int i=0; i < n; i++)
    {
        printf("%c", t);
```

[13] Tegenwoordig mag dat overal binnen een blok. En mag het ook binnen de `for`.

```
    }
}
```

De functie `tlijn()` kan zo opgeroepen worden:

```
tlijn('+',20);
tlijn('=',45);
```

Return en functietype

Als we een resultaat van een functie willen bekomen, dan wordt dit doorgegeven met de `return` opdracht. We moeten dan wel aangeven welke soort waarde met de return doorgegeven wordt. Daarom plaatsen we een type voor de functienaam. Dus niet alleen variabelen en constanten zijn van een bepaald type, ook functies worden met een type verbonden. Als we de functieoproep in een uitdrukking plaatsen, dan wordt de oproep vervangen door het resultaat van de functie.

```
int eigen_abs(int a) /* int : functietype */
{
   if (a < 0)
   {
      return -a;
   }
   else
   {
      return a;
   }
}

int main()
{
   int x,y,z;

   printf("geef 2 getallen:");
   scanf("%d %d",&x,&y);
   z = eigen_abs(x) + eigen_abs(y);
   printf("%d\n", z);

   return 0;
}
```

Het functietype mag niet weggelaten worden[14]. Als we helemaal geen resultaat willen teruggeven, dan moet dit expliciet aangegeven worden met het woord `void` (leeg). Hetzelfde kunnen we doen als een functie geen parameters ontvangt. We plaatsen dan niets tussen de functiehaken. Een functie die geen parameters ontvangt en geen resultaat geeft schrijven we zo:

```
void doe()
{

}
```

We geven nu nog een voorbeeld met een ander functietype.

```
float gemiddelde(float a, float b, float c)
{
   return (a + b + c)/3;
}
```

Je mag een `return` ook in het midden van een functie plaatsten. Als deze `return` uitgevoerd wordt, dan wordt de functie ogenblikkelijk verlaten. Dit is handig om te vermijden dat de rest van de functie uitgevoerd wordt.

[14] Vroeger mocht dat wel. Dan werd er `int` verondersteld.

```
void deel(int a, int b)
{
   printf("probeer te delen\n");

   if (b == 0)
   {
      return;
   }

   printf("gehele deling %d\n", a/b);
}
```

Het bovenstaande voorbeeld gaat de deling alleen maar uitvoeren als de deler verschillend van nul is. Omdat de functie in dit voorbeeld geen waarde teruggeeft, zie je ook geen uitdrukking achter het woord `return`.

We moeten hier toch nog zeggen dat het niet mogelijk is om een functie een doorgegeven variabele te laten wijzigen.

```
void verhoog(int a)
{
   a++;
}

void main()
{
   int b = 1;

   verhoog(b);
}
```

Omdat de functie `verhoog()` met een kopie van `b` werkt, wordt alleen `a` verhoogd. De variabele `b` blijft hier ongewijzigd. Men spreekt in dit geval van waardeparameter. In het Engels wordt dit aangegeven met de term *call by value*.

In C worden de parameters altijd met *call by value* doorgegeven. Als je hiervan wil afwijken in C, kan je niet anders dan pointers te gebruiken bij parameters. Om dat te verduidelijken wordt in de volgende sectie de adresoperator uitgelegd.

De & operator

De `&` operator bij een variabelenaam geeft het adres van die variabele. We kunnen nagaan waar een variabele zich in het geheugen bevindt.

```
v = 12;
printf("het getal %d staat in adres %u\n",
                    v, &amp;v);
```

Resultaat:

```
het getal 12 staat in adres 65502
```

Met het volgende voorbeeld zien we dat twee variabelen met dezelfde naam een verschillend adres hebben. Het zijn dus verschillende variabelen.

```
void fu()
{
   int a = 7;

   printf("fu: a = %d &a = %u\n", a, &a);
}

int main()
```

```
{
   int a = 5;

   printf("main: a = %d &a = %u\n", a, &a);
   fu();

   return 0;
}
```

Resultaat:

```
main: a = 5 &a = 65502
fu: a = 7 &a = 65496
```

Pointers en adresparameters

De volgende functie is bedoeld om de inhoud van twee variabelen te verwisselen. Deze versie is niet correct omdat alleen de kopies van de doorgegeven variabelen verwisseld worden en niet de originelen.

```
void verwissel(int u, int v)
{
   int   help;

   help = u;
   u = v;
   v = help;
}

int main()
{
   int   x = 3, y = 4;

   printf("x: %d, y %d\n", x, y);
   verwissel(x,y);
   printf("x: %d, y %d\n", x, y);

   return 0;
}
```

De variabelen `x` en `y` blijven dus ongewijzigd. We kunnen hier ook geen return gebruiken omdat deze slechts 1 waarde teruggeeft. De oplossing is als volgt: we geven als actuele parameters niet de inhoud van `x` en `y` door, maar wel de adressen van `x` en `y`. Dit kunnen we doen met de adresoperator. Dit betekent dan wel dat we als formele parameters in de functie `verwissel()` variabelen moeten voorzien, die in staat zijn om adressen op te slaan. Deze soort variabelen noemt men *pointers*.

Vooraleer we pointers uitleggen, verklaren we eerst de declaratie van een gewone variabele. Bij de declaratie

```
int   getal = 123;
```

is getal van het type `int` en is `&getal` het adres van deze variabele.

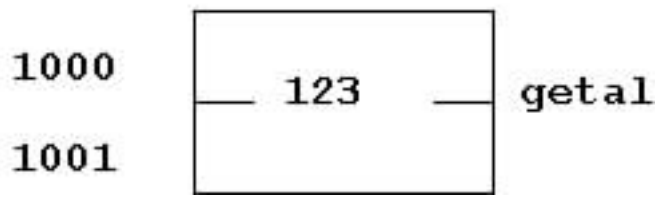

Figure 2: Een variabele in het geheugen

De inhoud van getal is 123 en het adres van getal is 1000. De uitdrukking `&getal` is van het pointertype en stelt een constante voor. We kunnen deze constante toekennen aan een pointervariabele:

```
ptr = &getal;
```

Dit wil zeggen dat `ptr` moet gedeclareerd worden als een pointervariabele.

```
int *ptr;
```

Dit wordt zo gelezen: `ptr` is een pointer naar een `int`. De operator `*` betekent hier pointer[15]. De variabele `ptr` kan als volgt gebruikt worden:

```
ptr = &getal;
a = *ptr;
```

De eerste opdracht plaatst het adres van getal in `ptr`. De tweede opdracht neemt de inhoud van de `int` variabele die aangewezen wordt door `ptr` en plaatst deze waarde in `a`. De variabele `a` krijgt dus de waarde van `getal`. De `*` operator is hier de operator voor indirecte verwijzing[16].

De situatie van deze variabelen kan zo weergegeven worden:

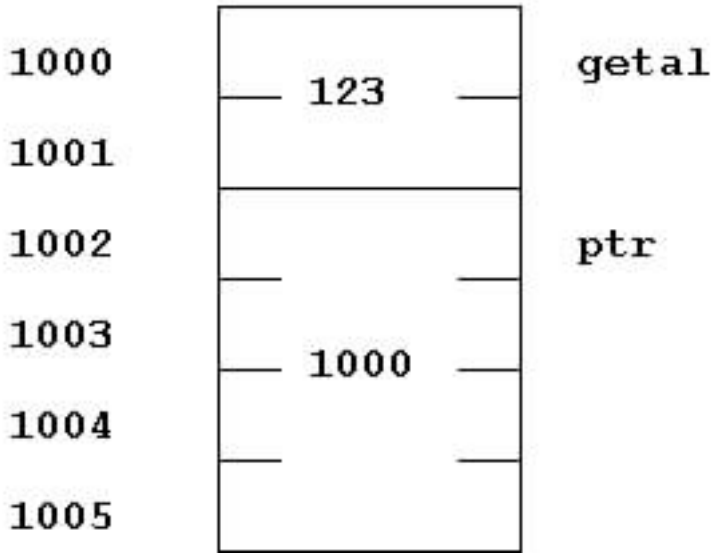

Figure 3: Een pointer in het geheugen

Bij de declaratie wordt vastgelegd dat `ptr` een pointer naar `int` is. We kunnen dus wel het adres van een `int` variabele in `ptr` plaatsen maar niet het adres van een `char` variabele.

De functie `verwissel()` is nu herschreven met pointers als formele parameter:

```
void verwissel(int *u, int *v)
{
   int   help;

   help = *u;
   *u = *v;
   *v = help;
}
```

[15] Opnieuw is dit een voorbeeld van dubbel gebruik van een teken, een `*` kan wijzen op een vermenigvuldiging of een declaratie van een pointer.

[16] De operator voor indirecte verwijzing bij een pointer is de derde betekenis van het teken `*`.

```
int main()
{
   int   x = 3, y = 4;

   verwissel(&x,&y);
   printf("x: %d, y %d\n", x, y);

   return 0;
}
```

Wanneer `verwissel()` opgeroepen wordt, krijgt de variabele `u` als inhoud het adres van `x` en `v` het adres van `y`. De inhoud van deze twee aangewezen variabelen wordt dan verwisseld.

We kunnen parameters als volgt samenvatten. Als we informatie doorgeven, kunnen we de inhoud van die variabele doorgeven:

```
// waarde:
int x;

fun1( x );
```

Ofwel kunnen we het adres van die variabele doorgeven:

```
// adres:
int x;

fun2( &x );
```

In het eerste geval wordt de waarde van de variabele doorgegeven en is er sprake van *call by value*. In het tweede geval wordt het adres van de variabele doorgegeven en is er sprake van *call by reference*. Merk op dat in het tweede geval (adres doorgeven) het mogelijk is om in de functie de inhoud van de variabele, waarvan het adres is doorgegeven, te wijzigen.

Tot slot hernemen we nog het foutieve voorbeeld uit de inleiding.

```
int *p;
*p = 17;
```

In dit fragment zie eerst de declaratie van de variabele `p`. Hierbij is het type `int *`, dus een pointer naar een `int`. Deze variabele `p` wordt niet geïnitialiseerd; zijn waarde is onbepaald. De tweede regel toont een toekenning waarbij `17` weggeschreven wordt op de geheugenplaats die door de pointer `p` wordt aangewezen. Vermits de pointer als inhoud geen bepaalde adreswaarde heeft gekregen, kan je dus ook niet voorspellen waar de `17` in het geheugen terecht komt. Bijna altijd wordt het programma waarin dit fragment uitgevoerd wordt, met een foutmelding afgebroken.

Een mogelijke verbetering is de volgende:

```
int getal;
int *p = &getal;
*p = 17;
```

In dit geval wordt de pointer `p` geïnitialiseerd met het adres van de variabele `getal` en bijgevolg komt de waarde `17` terecht in de variabele `getal`.

`inline` functies

Met het woord `inline` kan je ervoor zorgen dat er bij de oproep van een functie de code van deze functie rechtstreeks uitgevoerd wordt zonder subroutine call en return instructie. Voor zeer korte functies win je dan een beetje snelheid zonder dat de code veel langer wordt. `inline` functies maken zo kans om macro's te vervangen.

Hier is een voorbeeld:

```
static inline int f4(int a, int b)
```

```
{
   return a + b;
}

int f5()
{
   int som = f4(3, 4);
   return som;
}
```

De functie `f4()` is `inline` gemaakt. De woord `static` moet je erbij zetten om te vermijden dat de functie `f4()` van buiten de huidige module gebruikt kan worden. Het gebruik van een `inline` functie is beperkt tot de huidige module omdat er voor dit type functie geen adres in de code wordt vastgelegd. En daardoor zal de linker externe verwijzingen naar een `inline` functie niet kunnen oplossen.

Recursie

Functies in C hebben eigenschappen die het gemakkelijk maken om recursie[17] toe te passen bij het oplossen van problemen. Zo worden gegevens naar de functie doorgegeven via parameter en krijg je één resultaat terug met `return`. Recursie betekent dat een functie zichzelf oproept. Hier is een eerste voorbeeld:

```
#include <stdio.h>

void toon(int n)
{
   if (n > 0)
   {
      printf("%d\n", n);
      toon(n - 1);
   }
}

int main()
{
   toon(5);
}
```

De uitvoer van dit programma is:

```
5
4
3
2
1
```

In dit voorbeeld worden alle getallen van `n` omlaag tot en met 1 getoond. Deze techniek is mogelijk omdat bij elke oproep van de functie er een nieuwe kopie van de parameter `n` wordt gemaakt. De toestand van elke oproep wordt dus bewaard. `toon(5)` wordt gestart en toont een 5, daarna wordt `toon(4)` gestart en wordt het getal 4 getoond. De `if` is belangrijk: van zodra we aan nul komen wordt de recursie gestopt. Als de `if` er niet zou staan, zouden we een oneindige recursie krijgen die uiteindelijk in *stack overflow* zal resulteren. Waarom krijgen we een stackprobleem? Omdat de toestand van de oproep van een functie (parameters en lokale variabelen) op de stack wordt bijgehouden.

Meestal gebruik je recursie omdat er een recursieve definitie van het probleem bestaat. Voor dit probleem kan je zeggen dat het tonen van een reeks getallen bestaat uit het tonen van het eerste getal gevolgd door het tonen van de rest van de lijst. Hier is een variante van de `toon()` functie:

```
void toon2(int n)
{
   if (n > 0)
```

[17] Recursie is niet specifiek voor C en kan je in de meeste andere programmeertalen toepassen. Recursie in C is mogelijk omdat de parameters, lokale variabelen en het returnadres op de stack worden bijgehouden.

```
   {
      toon2(n - 1);
      printf("%d\n", n);
   }
}
```

In deze variante is de volgorde van de acties binnen de `if` omgekeerd. Dit geeft een lijst van 1 tot en met `n`, de volgorde is nu omgekeerd.

Dit is nu de uitvoer van dit programma:

```
1
2
3
4
5
```

Hier is de recursieve definitie van de berekening van de grootste gemene deler:

```
ggd(a, 0) => a
ggd(a, b) => ggd(b, a modulo b)
```

In woorden betekent dit dat de grootste gemene deler van twee getallen het eerste getal is als het tweede getal nul is. Wanneer dit niet zo is, kan je het getallenpaar `a, b` vervangen door `b, a modulo b`. Modulo is hier de gehele deling waarbij je de rest van de deling overhoudt. Als we dit in C omzetten, dan krijgen we dit:

```
// met a >= b
int ggd(int a, int b)
{
   if (b == 0)
   {
      return a;
   }
   else
   {
      return ggd(b, a % b);
   }
}
```

Als we de oproep `ggd(24, 18)` starten, dan is dit de sequentie van oproepen:

```
ggd(24, 18)
ggd(18, 6)
ggd(6, 0)
```

Dit algoritme heeft als voordeel dat het relatief snel een antwoord geeft. En deze snelheid krijgen we niet altijd bij recursie. Het volgende voorbeeld berekent het n-de getal van de reeks van Fibonacci.

```
int fib(int n)
{
   if (n == 0 || n == 1)
   {
      return 1;
   }
   else
   {
      return fib(n - 1) + fib(n - 2);
   }
}
```

Zoals je ziet, krijg je bij dit probleem dubbelrecursie: de functie roept zijn eigen tweemaal op. Dit is op zich niet erg maar in dit voorbeeld krijgen we wel heel veel oproepen. Als we bijvoorbeeld starten met `fib(5)`, dan is dit het overzicht van alle oproepen.

```
fib(5)-------------------------+
 |                              |
fib(4)------------+            fib(3)-----+
 |                |             |         |
fib(3)-----+     fib(2)--+     fib(2)--+ fib(1)
 |         |      |      |      |      |
fib(2)--+ fib(1) fib(1) fib(0) fib(1) fib(0)
 |      |
fib(1) fib(0)
```

Tot slot is er nog een interessante vaststelling die je kan maken als je de werking van recursie analyseert. Je ziet dat in alle bovenstaande voorbeelden geen enkele variabele na de initialisatie nog gewijzigd wordt. Deze opmerkelijke eigenschap maakt dat recursie dikwijls de aangewezen programmeerstijl is als je functionele programmeertalen[18] gebruikt. In een aantal van die talen is het niet mogelijk om na de initialisatie een variabele nog te wijzigen. Een voordeel hiervan is dat er gegarandeerd geen zijeffecten zijn. Hierdoor is het gemakkelijker om verschillende delen van een programma (threads, functies) automatisch te verdelen over de verschillende cores die in de CPU aanwezig zijn.

Geheugenklassen

Elke variabele in een C programma behoort tot een geheugenklasse. Deze klasse bepaalt de levensduur en de bereikbaarheid van de variabele. Voor elke variabele kiezen we een gepaste klasse.

De klasse waartoe een variabele behoort, kunnen we bepalen met een sleutelwoord bij de declaratie. De volgende sleutelwoorden worden hier besproken: `auto`, `extern`, `static`, `register` en `volatile`. Eén van deze woorden kan voor het type geplaatst worden bij een declaratie.

geheugenklasse + type + variabelenaam

Automatische variabelen

Dit zijn alle variabelen binnen een functie. We kunnen deze variabelen ook aanduiden met de term lokale variabelen. De ruimte voor deze variabelen en ook voor de formele parameters wordt gereserveerd op de stack. Vermits de stack een beperkte geheugenruimte omvat, moeten we de hoeveelheid lokale variabelen beperken.

```
void fu()
{
   int klad;

   klad = 1;
}
```

Deze variabelen bestaan alleen tijdens de uitvoering van de functie. Dit betekent dat er bij de start van de functie geheugen wordt gereserveerd voor de automatische variabelen. Dit geheugen wordt terug vrijgegeven bij het verlaten van de functie. We zouden het woord `auto` kunnen gebruiken, maar dit wordt altijd weggelaten[19]. Variabelen binnen een functie gedeclareerd zonder een geheugenklasse zijn altijd automatisch of lokaal.

Het is duidelijk dat we geen lokale variabele kunnen gebruiken voor gegevens op te slaan die tijdens de hele uitvoering van het programma moeten blijven bestaan.

Externe variabelen

De term `extern` wordt bij gebruikt voor de globale variabelen. Hiermee bedoelen we de variabelen die buiten de functies gedeclareerd worden.

Het woord `extern` kan bij een declaratie buiten een functie voorkomen. We hebben hier te maken met een verwijzing en geen geheugenreservatie.

[18] Talen zoals Clojure, Erlang, F#, Haskell, Lisp en gedeeltelijk ook Scala.
[19] Ondertussen heeft `auto` een nieuwe betekenis gekregen in C++11.

```
extern int waarde;   // geen geheugen allocatie

void fu()
{
   waarde = 3;
}
```

Hier wordt aangegeven dat de variabele `waarde` in een ander bestand gedeclareerd is. In C kunnen we met meerdere programmabestanden werken die gemeenschappelijke variabelen hebben.

`static` variabele

Hiermee bedoelen we variabelen die altijd bestaan, ook al staat de declaratie binnen een functie. De externe variabelen zijn statisch omdat ze altijd bestaan tijdens de levensduur van het programma.

Gebruik `static` binnen functie

We geven een voorbeeld.

```
void probeer()
{
   int  tijdelijk = 1;
   static int altijd = 1;

   printf("tijdelijk %d, altijd %d\n",
   tijdelijk++, altijd++);
}

int main()
{
   for (int i=1; i < 10; i++)
   {
      printf("%d :", i);
      probeer();
   }

   return 0;
}
```

De functie `probeer()` heeft twee variabelen `tijdelijk` en `altijd`. De variabele `tijdelijk` is automatisch, ze bestaat enkel tijdens de uitvoering van `probeer()`. De variabele `altijd` is statisch en bestaat tijdens de hele uitvoering van het programma. De variabele `tijdelijk` krijgt de initialisatiewaarde bij elke oproep van `probeer()`. De variabele `altijd` wordt slechts éénmaal geïnitialiseerd, namelijk bij de start van het programma.

Het woord `static` maakt van een tijdelijke variabele een variabele die altijd bestaat. Dit kan soms handig zijn, maar het kan ook ongewenste zijeffecten leveren[20].

Gebruik `static` buiten functie

Hiermee creëren we een externe variabele die enkel bekend is binnen het bestand. Het volgende voorbeeld maakt dit duidelijk.

bestand 1

```
#include <header.h>

int   a;
static int b;

static void fu1()
```

[20] Merk op dat `static` in C++ een totaal andere betekenis heeft. Toch kan `static` in beide betekenissen in gebruik blijven.

```
{
   fu2();
}

int main()
{
   fu1();
   fu3();

   return 0;
}
```

bestand 2

```
#include <header.h>

void fu3()
{
   printf("%d\n",a);
}

void fu2()
{
   fu3();
}
```

Met behulp van de `#include` aanwijzing wordt het bestand `header.h` ingelast in bestand 1 en bestand 2. Deze bevat de volgende tekst:

```
void fu2();
void fu3();

extern int a;
```

Dit zijn aanwijzingen hoe de functies `fu2()`, `fu3()` en de variabele `a` gebruikt moeten worden. De notatie voor de functies noemt men een functieprototype. Hierdoor is het mogelijk dat de compiler een foutmelding geeft als een functie uit een ander bestand, verkeerd opgeroepen wordt. De prototypes worden ook ingelast in het bestand waar de functies vastgelegd worden. Hierdoor wordt gegarandeerd dat de prototypes precies overeenstemmen met de functies zelf. De twee bestanden worden afzonderlijk gecompileerd en daarna samengevoegd in de linkfase.

In het voorbeeld is de variabele `a` is bekend in `main()`, `fu1()`, `fu2()` en `fu3()`. De variabele `b` is alleen bekend in `main()` en `fu1()`.

Het woord `volatile` wordt o.a. gebruikt om aan te geven dat een variabele door een interruptroutine gewijzigd kan worden. Hierdoor weet de compiler dat er geen optimalisatie mag gebeuren die de waarde van een gewijzigde variabele voor langere tijd in een processorregister bijhoudt. In normale omstandigheden heb je het woord `volatile` niet nodig.

Tot slot geven we nog een overzicht dat al de geheugenklassen weergeeft.

	soort klasse	woord	levensduur	bereik
binnen functie	automatisch	parse_error	tijdelijk	lokaal
	register	parse_error	tijdelijk	lokaal
	statisch	parse_error	altijd	lokaal
buiten functie	extern	parse_error	altijd	in alle bestanden
	extern static	parse_error	altijd	in 1 bestand

Arrays en pointers

Arrays zijn variabelen die meerdere waarden van een zelfde soort kunnen opslaan. Pointers zijn verwijzingen naar andere variabelen. We behandelen eerst arrays en daarna het verband met pointers.

Array voorbeelden

```
int getal[10];
float r[100];
char t[20];
```

Elk van deze variabelen is een array. De array `getal` bevat 10 elementen:

```
getal[0], getal[1], ... , getal[9]
```

De index die gebruikt wordt om de elementen te bereiken, start bij 0 en loopt tot het aantal elementen - 1. Het volgende voorbeeld toont hoe arrays gebruikt kunnen worden.

```
#define DIM 10

int main()
{
   int som, getallen[DIM];

   for (int i=0; i<DIM; i++)
   {
      scanf("%d",&getallen[i]);
   }

   printf("dit zijn de getallen\n");
   for (int i=0; i<DIM; i++)
   {
      printf("%5d",getallen[i]);
   }
   printf("\n");

   for (int i=0, som=0; i<DIM; i++)
   {
      som += getallen[i];
   }
   printf("het gemiddelde is %d\n",som/DIM);
}
```

Initialisatie van arrays

Net zoals enkelvoudige variabelen kunnen ook arrays geïnitialiseerd worden. Dit kan zowel bij externe en statische arrays en ook bij arrays als lokale variabelen.

```
// dagen per maand
int dagen[12] = {31,28,31,30,31,30,31,31,30,31,30,31};

int main()
{
   for (int i=0; i<12; i++)
   {
      printf("%d dagen in maand %d\n",dagen[i],i+1);
   }

   return 0;
}
```

De waarden waarmee de array gevuld wordt, worden tussen accolades geplaatst. Indien er te weinig waarden zijn, dan worden de laatste elementen van de array met 0 gevuld.

Hier is een andere versie:

```
// dagen per maand
int dagen[] = {31,28,31,30,31,30,31,31,30,31,30,31};

int main()
{
   for (int i=0; i<sizeof(dagen)/sizeof(int); i++)
   {
      printf("%d dagen in maand %d\n",dagen[i],i+1);
   }
   return 0;
}
```

In deze versie is de lengte van de array weggelaten. De lengte wordt nu bepaald door het aantal getallen tussen accolades. De lengte mag alleen maar weggelaten worden als de array geïnitialiseerd wordt.

Verband tussen pointers en arrays

De arraynaam is een pointer naar eerste element. Dit verband verduidelijken we met een voorbeeld.

```
int rij[20];
```

Bij deze array is `rij[0]` het eerste element. Het adres hiervan is `&rij[0]`. Dit kan ook korter geschreven worden: `rij` en `&rij[0]` zijn hetzelfde. Ze duiden allebei het adres van de array aan. Het zijn allebei pointerconstanten.

In het volgende voorbeeld wordt er met pointers gerekend.

```
int main()
{
   int getallen[4], *pget;
   char tekens[4], *ptek;

   pget = getallen;
   ptek = tekens;
   for (int i=0; i<4; i++)
   {
      printf("pointers + %d: %u %u\n",
         i, pget + i, ptek + i);
   }
   return 0;
}
```

De eerste toekenning plaatst het adres van de array getallen in de pointervariabele `pget`. De tweede toekenning doet een gelijkaardige bewerking. In de `printf()` opdracht wordt de lusteller `i` opgeteld bij de inhoud van de pointers. Dit resultaat komt op het scherm:

```
pointers + 065486 65498
pointers + 165488 65499
pointers + 265490 65500
pointers + 365492 65501
```

De eerste regel geeft de adressen van de eerste elementen van de arrays. De volgende regel geeft de adressen van de tweede elementen enzovoort. We zien dus het volgende: als we de inhoud van de pointer verhogen met 1, dan wordt het adres, dat in de pointervariabele wordt opgeslagen, verhoogd met de breedte van het aangeduide element. De pointer `pget` wijst naar `int`, `int` is 2 bytes breed[21] dus wordt er 2 opgeteld bij de inhoud van `pget`. Dezelfde regel kunnen we toepassen voor de pointer `ptek`. Die wordt verhoogd met 1 (breedte `char`).

[21] Dit voorbeeld stamt nog uit het tijdperk van de 16 bit besturingssystemen, toen was een `int` nog 16 bit.

De array `getal` kan zo voorgesteld worden:

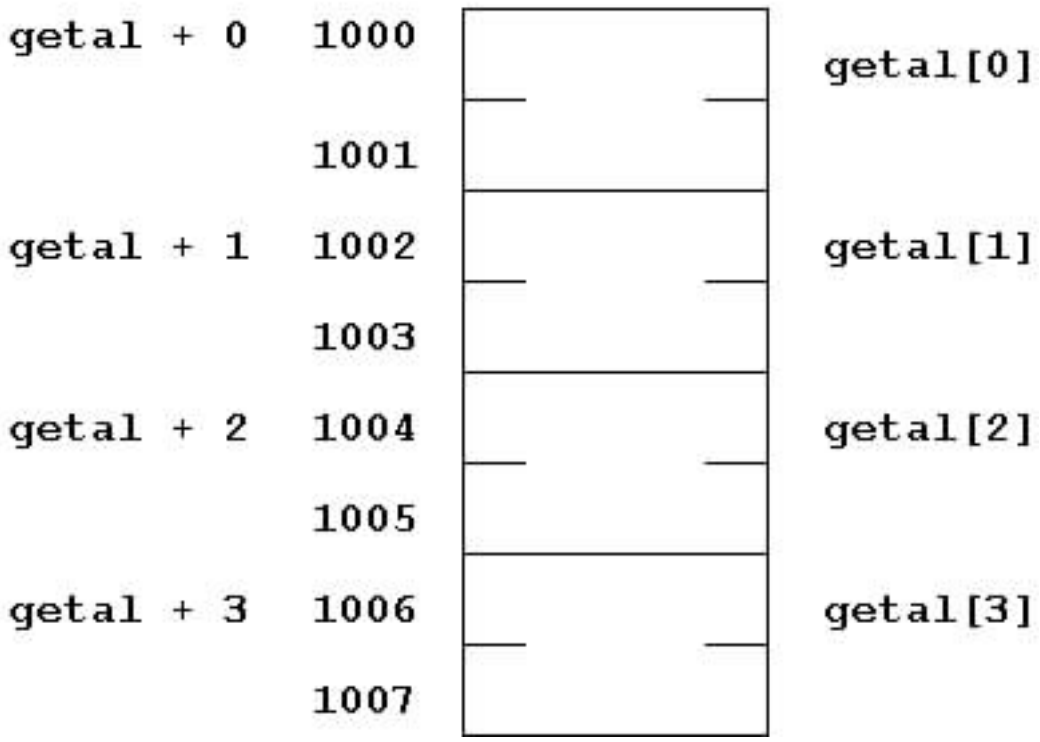

Figure 4: Layout van een array in het geheugen

`getal + 2` en `&getal[2]` stellen beide hetzelfde adres voor.

`*(getal + 2)` en `getal[2]` stellen beide dezelfde waarde voor.

Opgelet: `*getal + 2` is de waarde van het eerste element verhoogd met 2. Deze uitdrukking is dus niet hetzelfde als `*(getal + 2)`. De haken zijn nodig omdat `*` een hogere prioriteit heeft dan `+`.

Hetzelfde probleem ontstaat bij de interpretatie van `*p++` . Is dit `(*p)++` of `*(p++)`? Het antwoord is de tweede uitdrukking omdat `*` en `++` dezelfde prioriteit hebben en unaire operatoren van rechts naar links groeperen.

Arrays als functieparameters

Als formele parameter kunnen we arrays gebruiken. De afmeting van de array mag weggelaten worden.

```
void druk(int rij[])
{

}

int main()
{
   int reeks[50];

   druk(reeks);

   return 0;
}
```

Bij formele parameters de schrijfwijze `int rij[]` is `rij` een pointer variabele, geen array variabele. We

geven hier niet de inhoud van de array door, maar wel het adres. Dus `int rij[]` en `int *rij` zijn hetzelfde als formele parameter.

We kunnen de lengte van de array doorgeven:

```
void druk(int rij[], int lengte)
{
   for (int i=0; i<lengte; i++)
   {
      printf("%d\n", rij[i]);
   }
}
```

Deze versie doet identiek hetzelfde, alleen de toegang tot de array is gewijzigd:

```
void druk(int rij[], int lengte)
{
   for (int i=0; i<lengte; i++)
   {
      printf("%d\n", *(rij + i) );
   }
}
```

En tot slot de (mogelijk) snelste versie:

```
void druk(int rij[], int lengte)
{
   int *p;

   p = rij;               // bew 1
   peinde = &rij[lengte];
   while (p < &rij[lengte])  // bew 5
   {
      printf("%d\n", *p );// bew 2
      p++;                // bew 4
   }
}
```

In deze laatste versie wordt de pointervariabele `p` gebruikt om de elementen van de array te bereiken. Met de `++` operator wijst `p` telkens naar het volgende element in de array. De herhaling stopt als `p` het eerste adres aanwijst dat niet tot de array behoort.

Deze versie is ook de variant die geschreven zou kunnen worden door iemand die een sterke assembler achtergrond heeft. Merk ook op dat in de voorwaarde van de `while` de uitdrukking `&rij[lengte]` voorkomt. Dit is een uitdrukking die een constante waarde oplevert. Opmerkelijk is hier dat de index tussen de haken net iets te groot. Deze index levert het adres op van de eerste byte die geen deel meer uitmaakt van de array. Dit soort uitdrukkingen is toegelaten in C.

Hier is een samenvatting van de pointerbewerkingen:

1. toekenning

   Een adres wordt toegekend aan een pointervariabele.

2. waarde ophalen

   De `*` bewerking vindt de waarde die door de pointer wordt aangeduid.

3. een pointeradres nemen

   De pointer `int *p` bevindt zich op adres `&p`. Dit kan dienen als actuele parameter voor een functie die de doorgegeven pointer wijzigt.

4. een pointer verhogen

   Na deze bewerking wijst de pointer naar het volgende element.

5. het verschil tussen 2 pointers

   Dit geeft het aantal elementen dat zich tussen de 2 aangeduide posities bevindt.

Merk tenslotte op dat een functie een doorgegeven array kan wijzigen, juist omdat een array als een pointer wordt doorgegeven.

Arrays met meerdere dimensies

Bij de declaratie plaatsen we meerdere indexen na de arraynaam. Elke index staat apart tussen de rechte haken.

```
double matrix[3][4];
```

In het volgende voorbeeld zien we de initialisatie en het gebruik van een meerdimensionele array.

```
#include <stdio.h>

int main()
{
   static double matrix[3][4] =
   {
      { 2,5,9,7 },
      { 8,1,3,4 },
      { 10,5,45,23 }
   };

   for (int i=0; i<3; i++)
   {
      for (int j=0; j<4; j++)
      {
         printf("%5.2f ", matrix[i][j] );
      }
      printf("\n");
   }
   return 0;
}
```

De variabele `matrix` kunnen we voorstellen als een matrix die bestaat uit 3 rijen met elk 4 elementen. De variabele wordt rij per rij geïnitialiseerd (alleen statische en externe arrays kunnen geïnitialiseerd worden). De getallen 2, 5, 9 en 7 komen terecht in de eerste rij. We kunnen ze terugvinden in de elementen `matrix[0][0]`, `matrix[0][1]`, `matrix[0][2]` en `matrix[0][3]`. Op dezelfde wijze worden de twee andere rijen gevuld. Het is ook mogelijk om de binnenste paren accolades, die telkens een rij getallen afsluiten, weg te laten. Dit is identiek in werking maar is minder overzichtelijk.

Pointers naar functies

Zoals reeds vermeld moet bij de declaratie van een pointer aangegeven worden naar welk type deze pointer wijst. We kunnen in de taal C ook een functietype gebruiken als het aangewezen type. We geven een voorbeeld:

```
int (*pf)(int a,int b);
```

De variabele `pf` is een pointer die wijst naar een functie die 2 `int` verwacht en die een `int` als resultaat teruggeeft. Deze variabele krijgt met een toekenning een waarde.

```
int fu(int a,int b)
{
   return( a + b);
}

pf = fu;
```

De pointer `pf` krijgt als waarde het adres van de functie `fu`. We kunnen de aangewezen functie oproepen via de pointervariabele.

```
c = (*pf)(1,2);
```

De mogelijkheid in C om pointers naar functies te maken wordt niet veel gebruikt, maar in C++ duikt ze opnieuw op in de vorm van lambdafuncties.

Dit zijn een aantal extra oefeningen over pointers en arrays.

1. Maak een functie die het gemiddelde berekent van een array met getallen. Als parameter geef je de array en het aantal getallen door.

   ```
   double gemiddelde(double *p, int n);
   ```

2. Sorteer een array getallen met de bubble sort methode. Bij dit algoritme loop je door de lijst en verwissel je twee naburige getallen als ze niet in de juiste volgorde staan. Dit blijf je herhalen to er geen wisselingen meer zijn. Plaats de sorteeralgoritme in een eigen functie.

   ```
   void sorteer(int *p, int n);
   ```

3. Maak een functie die een getal zoekt in een array van getallen. Als het getal gevonden wordt, wordt de index van het getal teruggegeven (`0 .. n-1`); als het getal niet gevonden wordt, wordt -1 teruggegeven.

   ```
   int zoek(int *p, int n, int getal);
   ```

4. Maak een functie die een string (array van `char`) kopieert. Denk eraan dat je het einde van de string kan herkennen aan de binaire nul.

   ```
   void kopieer(char *naar, char *van);
   ```

5. Maak een functie die een string omkeert.

6. Maak een functie die de woorden in string omkeert. De woorden blijven op hun plaats staan maar de letters ervan worden omgekeerd. Tussen de woorden staan één of meerdere spaties.

7. Maak een functie die nagaat of een string een palindroom is. Een palindroom is een woord dat hetzelfde blijft als je het omkeert.

8. Maak een functie die een string als parameter krijgt en deze string omzet naar de p-taal. Dit betekent dat je na elke klinker a, o, e, i en u een letter p gevolgd door dezelfde klinker bijvoegt. Denk eraan dat de nieuwe string langer wordt dan het origineel. Hou hiermee rekening. Geef de nieuwe string met return terug.

9. Maak een functie die 2 strings vergelijkt. De strings hoeven niet even lang te zijn. Bij gelijkheid geef je 0 terug, bij ongelijkheid een andere waarde.

   ```
   int vergelijk(char *p, char *q);
   ```

10. Maak een functie die 2 arrays van gesorteerde getallen samenvoegt in een nieuwe array.

11. Maak een functie die een getal zoekt in een array van getallen. Deze array is gesorteerd in stijgende volgorde. Als het getal gevonden wordt, wordt de index van het getal teruggegeven (`0 .. n-1`); als het getal niet gevonden wordt, wordt -1 teruggegeven. Maak de oplossing recursief volgens de binaire zoekmethode: als het getal in de eerste helft zit, ga je verder met de eerste helft. Anders zoek je verder in de tweede helft.

    ```
    int zoekbinair(int *p, int n, int getal);
    ```

12. Maak een functie die de waarde uitrekent van een reeks getallen en bewerkingen. Een getal bestaat maar uit één cijfer en als bewerkingen komen alleen maar `+` en `*` voor. Zo zal de string `"234+*"` als `(3 + 4)*2` uitgerekend worden.

    ```
    int reken(char *p);
    ```

13. Maak een functie die in een tekst op zoek gaat naar een patroon. In het patroon kan de `*` als jokerteken voorkomen. Ook letters en cijfers mogen in een patroon voorkomen. Deze oefening is niet eenvoudig.

```
int zoekpatroon(char *tekst, char *patroon);
```

Tekenstrings en stringfunctions

Strings definiëren

Een string is een opeenvolging van `char` constanten, waarbij het einde aangeduid wordt door 0. We kunnen een stringconstante samenstellen met 2 aanhalingstekens:

```
"dit is een string"
```

Deze constante heeft een dubbele functie: ze zorgt voor opslag van de tekens in het geheugen en ze fungeert als constante van het type pointer naar `char`. In het volgende voorbeeld wordt het adres van een stringconstante opgeslagen in een pointervariabele.

```
char *pstr;

pstr = "dit is een string";
printf("%s",pstr);
```

Via initialisatie wordt een stringconstante opgeslagen in een `char` arrayvariabele. Tussen de rechte haken hoeft geen afmeting vermeld te worden.

```
char str1[] = {'a','b','c','d','e','\0' };
```

ofwel

```
char str1[] = "abcde";
```

De lengte van array is 6: 5 tekens + 1 nul. Als we de naam `str1` gebruiken, dan is dit een pointer naar het eerste element. Zo kunnen we enkele gelijke uitdrukkingen opstellen:

`str1` en `&str1[0]`

`*str1` en `'a'`

`*(str1+2)` en `str[2]` en `'c'`

Er is een verschil tussen de array en de pointer declaratie, maar wel zijn het allebei geïnitialiseerde variabelen:

```
char *ptekst = "een";
char atekst[]= "twee";
```

De variabele `ptekst` is een pointer die geïnitialiseerd wordt met het adres van de string `"een"`. Deze string bevindt zich elders in het geheugen. De variabele atekst is een array die geïnitialiseerd wordt met de string `"twee"`. Dit betekent dat `atekst` plaats heeft voor 5 tekens. We kunnen de geheugenverdeling zo voorstellen:

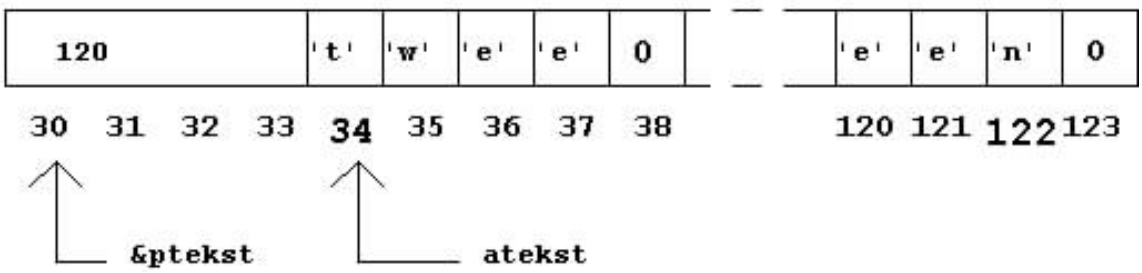

Figure 5: Strings in het geheugen

We hebben de volgende overeenkomsten:

```
&ptekst ---> 30
ptekst       120
*ptekst      'e'
ptekst[0]    'e'
atekst       34
*atekst      't'
atekst[0]    't'
```

`ptekst` is een pointervariabele en kan dus gewijzigd worden; `atekst` niet:

```
while ( *ptekst != 0)
{
   putchar ( *ptekst++ );
}
```

Deze herhaling drukt alle tekens van de string op het scherm.

`atekst` is een pointerconstante die wijst naar het eerste element van de array. `atekst` kan niet gewijzigd worden.

```
atekst++;  // FOUT
```

De inhoud van de array kan wel gewijzigd worden:

```
atekst[0] = 'p';
```

Arrays van tekenstrings

We declareren de volgende variabele:

```
char *kleuren[3] = { "wit",
                     "zwart", "azuurblauw" };
```

De variabele `kleuren` is een array van pointers die wijzen naar `char` elementen. De pointers zijn elk geïnitialiseerd met het adres van een string. De uitdrukkingen `kleuren[0]`, `kleuren[1]`, en `kleuren[2]` zijn de 3 pointers. Als we er een `*` bijplaatsen krijgen we: `*kleuren[0]` is de eerste letter van de eerste string. In kleuren worden alleen adressen opgeslagen; de strings zelf worden elders in het geheugen opgeslagen.

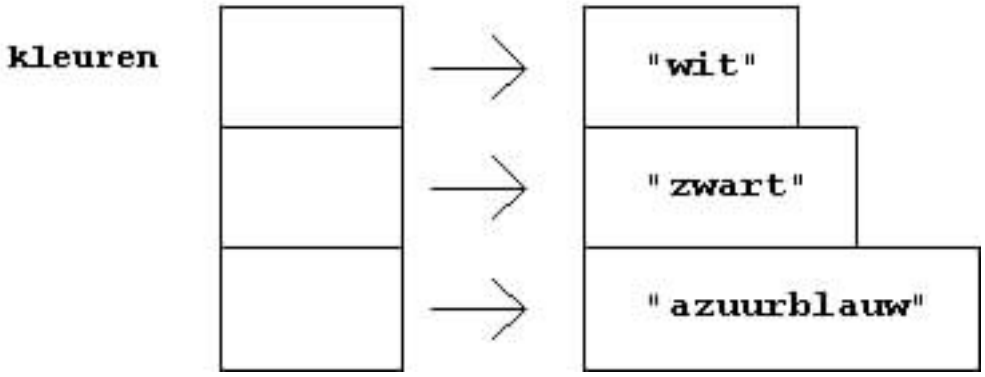

Figure 6: Array van pointers naar string

Deze variabele kan ook anders gedeclareerd worden. Het is nu een array met 2 dimensies. De strings worden in de array zelf opgeslagen. Voor de string `"wit"` betekent dit dat slechts een deel van de rij gebruikt wordt. Een deel van de array blijft dus onbenut.

```
char kleuren[3][11] = { "wit",
                     "zwart", "azuurblauw" };
```

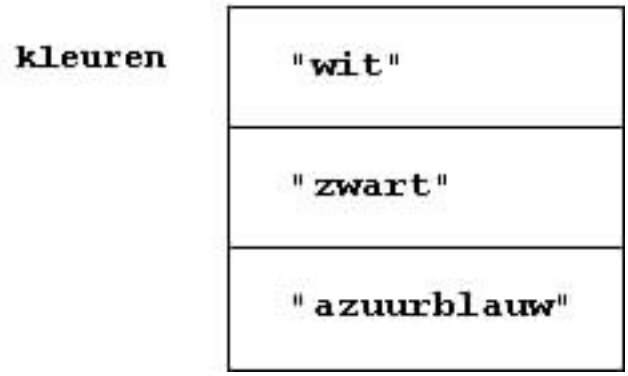

Figure 7: Arrays van strings

Stringin- en uitgave

We creëren eerst plaats voor de in te lezen string.

```
char *naam;

scanf("%s", naam);
```

Deze declaratie van `naam` levert een crash op; `naam` is een pointervariabele, die geen waarde gekregen heeft. `scanf()` gebruikt de inhoud van `naam` als adres om de ingelezen tekens op te slaan. Het resultaat is dus onvoorspelbaar. Een betere declaratie is dit:

```
char naam[81];
```

Stringin- en uitvoer doen we met de `gets()` en `puts()` functies.

```
int main()
{
   char naam[20][81]; /* plaats voor 20 namen */
   int  n;

   n = 0;
   while (gets(naam[n]) != NULL)
   {
      n++;
   }
   for (int i=0; i<n; i++)
   {
      puts(naam[i]);
   }
}
```

Het programma leest een aantal strings in met `gets()` en geeft daarna deze strings weer op het scherm. De functie `gets()` levert als resultaat het adres van de ingelezen string. Als de voorwaarde `EOF` (dit is end of file) voorkwam tijdens de ingave, is het resultaat 0. Deze eigenschap wordt in het programma gebruikt om de herhaling van de ingave stop te zetten. Let op de notatie `naam[n]`, dit is hetzelfde als `&naam[n][0]`.

Er zijn een aantal verschillen ten opzichte van `printf("%s")` en `scanf("%s")`. `gets()` leest alle ingegeven tekens in tot de return; de return zelf wordt niet opgenomen in de string. `scanf("%s")` start de string na de eerste whitespace (tab, newline of spatie) en stopt voor de volgende whitespace. Dit kan gebruikt worden om woorden uit een regel in te lezen. `puts()` doet altijd een newline op het einde van de string, `printf()` alleen als `\n` vermeld wordt.

Enkele stringfuncties

In C++ bestaat het `string` type met een hele reeks eigen methoden. Hierdoor zullen de volgende C functies minder belangrijk worden.

We bespreken enkele van de belangrijkste stringfuncties.

`strlen()`

Deze functie berekent de lengte van een string in bytes.

```
void pas(char *string, int lengte)
{
   if (lengte < strlen(string))
   {
      *(string + lengte) = '\0';
   }
}
```

De functie `pas()` kort een string in tot een gegeven lengte. Dit wordt gedaan door een 0 in de string bij te plaatsen.

`strcat()`

Deze functie voegt 2 strings samen.

```
int main()
{
   char naam[80];

   gets(naam);
   strcat(naam," is een mooie naam\n");
   puts(naam);

   return 0;
}
```

De functie `strcat()` ontvangt 2 `char` pointers. De string aangeduid door de eerste pointer wordt uitgebreid met de string aangeduid door de tweede pointer. De eerste string moet voldoende plaats hebben, anders worden andere variabelen overschreven.

Dit is meteen de zwakke plek van string in C: je moet altijd arrays maken die voldoende ruimte hebben om de verwachte grootte van de string te kunnen opslaan. Je mag bij een array nooit de maximale index overschrijden. Deze eis is heel hard en hieraan kan alleen maar voldaan worden als het programma correct geschreven is. Je hebt in C geen veiligheidsvangnet dat arrayindexfouten detecteert zoals in Java.

`strcmp()`

Deze functie vergelijkt 2 strings. Als de strings identiek zijn, is het resultaat 0, anders is het resultaat verschillend van 0.

```
int main()
{
   char antw[40];

   puts("waar woonde Julia ?");
   gets(antw);
   if (strcmp(antw, "Verona") == 0)
   {
      puts("goed");
   }
   else
   {
```

```
      puts("fout");
   }
   return 0;
}
```

`strcpy()`

Deze functie kopieert een string.

```
char kopie[40],*p;
p = "origineel";
strcpy(kopie,p);
```

In dit voorbeeld worden de letters van de string één voor één gekopieerd naar de array `kopie`.

Argumenten op de opdrachtregel

De argumenten die bij de programmastart worden doorgegeven, zijn bereikbaar vanuit het programma. Hiervoor wordt `main()` voorzien met twee formele parameters.

```
int main(int argc, char *argv[])
{
   for (int i=0; i<argc; i++)
   {
      printf("%s ",argv[i]);
   }
   printf("\n");

   return 0;
}
```

De eerste parameter `argc` geeft aan hoeveel argumenten er bij de programmastart meegegeven zijn. In dit aantal is de programmanaam meegerekend. De tweede parameter `argv` is een tabel van pointers naar `char`. Elke pointer wijst naar het eerste teken van een argumentstring. Deze strings zijn afgesloten met een 0. `argv[0]` wijst naar de programmanaam, `argv[1]` is het eerste argument, enzoverder. Het gebruik van een pointertabel laat een variabel aantal argumenten toe. De variabele `argc` geeft aan hoeveel pointers er in de tabel zitten. Je kan het voorbeeld ook anders schrijven door gebruik te maken van het feit dat het einde van de tabel ook aangegeven wordt door een pointer die `NULL` is.

```
int main(int argc, char *argv[])
{
   char **p = argv;
   while (*p != NULL)
   {
      printf("argument %s\n", *p);
      p++;
   }

   return 0;
}
```

Strings sorteren

Tot slot is hier nog een programmavoorbeeld, dat strings sorteert.

```
#include <stdio.h>
#include <string.h>

#define SLEN 81
#define DIM 20
#define STOP ""
```

```
void strsort(char *strings[], int num)
{
   char *temp;

   for (int klein=0; klein<num-1; klein++)
   {
      for (int zoek=klein+1; zoek<num; zoek++)
      {
         if ( strcmp(strings[klein],strings[zoek]) >0)
         {
            temp = strings[klein];
            strings[klein] = strings[zoek];
            strings[zoek] = temp;
         }
      }
   }
}

int main()
{
   static char ingave[DIM][SLEN]; // array voor ingave
   char *pstr[DIM];               // pointer tabel
   int tel = 0;

   printf("geef strings in\n");
   printf("eindig met een lege string\n");
   while( tel<DIM && gets(ingave[tel]) != NULL
          && strcmp(ingave[tel],STOP) != 0)
   {
      pstr[tel] = ingave[tel];
      tel++;
   }

   // sorteer met pointers
   strsort(pstr, tel);

   puts("\ndit is gesorteerd\n");
   for (int k=0; k<tel; k++)
   {
      puts(pstr[k]);
   }
}
```

Dit programma leest eerst een aantal strings in. De strings komen in de tweedimensionele array ingave terecht. De herhaling van de ingave stopt als er geen plaats meer is voor strings of als EOF optreedt of als er een lege string ingegeven wordt. Tijdens de ingave wordt de pointertabel `pstr` gevuld met het adres van elke string.

Met deze tabel `pstr` wordt het sorteren uitgevoerd. In plaats van strings te kopiëren (veel tekens kopiëren) worden er pointers gekopieerd. De pointertabel `pstr` wordt samen met het aantal strings doorgegeven aan de functie `strsort()`. Deze functie start bij de eerste string en gaat na of er verder nog strings zijn die kleiner zijn. Kleiner betekent hier: komt eerst in de alfabetische rangschikking. Hier wordt gebruik gemaakt van de eigenschap dat `strcmp()` iets zegt over de alfabetische volgorde als de 2 strings verschillend zijn. De mogelijke resultaten zijn:

```
strcmp("a", "a")    // geeft 0
strcmp("b", "a")    //       1 (positief)
strcmp("a", "b")    //      -1 (negatief)
```

Indien een kleinere string gevonden wordt, dan worden de pointers die wijzen naar de eerste en de gevonden

string verwisseld. Hetzelfde wordt herhaald voor de tweede tot en met de voorlaatste string.

Overzicht van de string functies

De prototypes van de functies voor stringmanipulatie zijn terug te vinden in de headerbestand `string.h`.

`strcpy()`

Kopieert string `src` naar `dest`.

Prototype:

```
char *strcpy(char *dest, const char *src);
```

Geeft `dest` terug.

```
#include <stdio.h>
#include <string.h>

int main()
{
   char string[10];
   char *str1 = "abcdefghi";

   strcpy(string, str1);
   printf("%s\n", string);
   return 0;
}
```

`strncpy()`

Kopieert maximum maxlen tekens van `src` naar `dest`.

Prototype:

```
char *strncpy(char *dest, const char *src, size_t maxlen);
```

Indien maxlen tekens gekopieerd worden, wordt geen nul teken achteraan bijgevoegd; de inhoud van dest is niet met een nul beëindigd.

Geeft `dest` terug.

```
#include <stdio.h>
#include <string.h>

int main()
{
   char string[10];
   char *str1 = "abcdefghi";

   strncpy(string, str1, 3);
   string[3] = '\0';
   printf("%s\n", string);
   return 0;
}
```

`strcat()`

Voegt `src` bij `dest`.

Prototype:

```
char *strcat(char *dest, const char *src);
```

Geeft `dest` terug.

```
#include <string.h>
#include <stdio.h>

int main()
{
   char destination[25];
   char *blank = " ";
   char *c = "C++";
   char *Borland = "Borland";

   strcpy(destination, Borland);
   strcat(destination, blank);
   strcat(destination, c);

   printf("%s\n", destination);

   return 0;
}
```

strncat()

Voegt maximum maxlen tekens van `src` bij `dest`.

Prototype:

```
char *strncat(char *dest, const char *src, size_t maxlen);
```

Geeft `dest` terug.

```
#include <string.h>
#include <stdio.h>

int main()
{
   char destination[25];
   char *source = " States";

   strcpy(destination, "United");
   strncat(destination, source, 7);
   printf("%s\n", destination);

   return 0;
}
```

strcmp()

Vergelijkt een string met een andere

Prototype:

```
int strcmp(const char *s1, const char *s2);
```

Geeft een waarde terug:

`< 0` indien `s1` kleiner dan `s2`

`== 0` indien `s1` gelijk is aan `s2`

`> 0` indien `s1` groter is dan `s2`

Voert een vergelijking met teken uit.

```
#include <string.h>
#include <stdio.h>
```

```
int main()
{
   char *buf1 = "aaa";
   char *buf2 = "bbb",
   char *buf3 = "ccc";
   int ptr;

   ptr = strcmp(buf2, buf1);
   if (ptr > 0)
   {
      printf("buffer 2 is greater than
                        buffer 1\n");    // ja
   }
   else
   {
      printf("buffer 2 is less than buffer 1\n");
   }
   ptr = strcmp(buf2, buf3);
   if (ptr > 0)
   {
      printf("buffer 2 is greater than
                        buffer 3\n");
   }
   else
   {
      printf("buffer 2 is less than buffer
                        3\n");       // ja
   }

   return 0;
}
```

`strncmp()`

Vergelijkt maximum maxlen tekens van de ene string met de andere.

Prototype:

```
int strncmp(const char *s1, const char *s2,
                        size_t maxlen);
```

Geeft een waarde terug:

`< 0` indien `s1` kleiner dan `s2`

`== 0` indien `s1` gelijk is aan `s2`

`> 0` indien `s1` groter is dan `s2`

Voert een vergelijking met teken (`signed char`) uit.

```
#include <string.h>
#include <stdio.h>

int main()
{
   char *buf1 = "aaabbb", *buf2 =
                        "bbbccc", *buf3 = "ccc";
   int ptr;

   ptr = strncmp(buf2,buf1,3);
   if (ptr > 0)
   {
```

```
      printf("buffer 2 is greater than
                        buffer 1\n");    // ja
   }
   else
   {
      printf("buffer 2 is less than buffer 1\n");
   }

   ptr = strncmp(buf2,buf3,3);
   if (ptr > 0)
   {
      printf("buffer 2 is greater than
                        buffer 3\n");
   }
   else
   {
      printf("buffer 2 is less than buffer
                        3\n");       // ja
   }
   return(0);
}
```

strchr()

Zoekt een teken `c` in `s`.

Prototype:

```
char *strchr(const char *s, int c);
```

Geeft een pointer terug naar de eerste plaats waar het teken `c` in `s` voorkomt; indien `c` niet voorkomt in `s`, geeft `strchr` `NULL` terug.

```
#include <string.h>
#include <stdio.h>

int main()
{
   char string[15];
   char *ptr, c = 'r';

   strcpy(string, "This is a string");
   ptr = strchr(string, c);
   if (ptr)
   {
      printf("The character %c is at
                        position: %d\n", c,
                  ptr-string); // 12
   }
   else
   {
      printf("The character was not found\n");
   }
   return 0;
}
```

strrchr()

Zoekt de laatste plaats waar `c` in `s` voorkomt.

Prototype:

```
char *strrchr(const char *s, int c);
```

Geeft een pointer terug naar de laatste plaats waar `c` voorkomt, of `NULL` indien `c` niet voorkomt in `s`.

```
#include <string.h>
#include <stdio.h>

int main()
{
   char string[15];
   char *ptr, c = 'i';

   strcpy(string, "This is a string");
   ptr = strrchr(string, c);
   if (ptr)
   {
      printf("The character %c is at
                        position: %d\n", c,
                  ptr-string); // 13
   }
   else
   {
      printf("The character was not found\n");
   }
   return 0;
}
```

strspn()

Doorzoekt een string naar een segment dat is een subset van een reeks tekens.

Prototype:

```
size_t strspn(const char *s1, const char *s2);
```

Geeft de lengte van het initiële segment van `s1` dat volledig bestaat uit tekens uit `s2`.

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>

int main()
{
   char *string1 = "1234567890";
   char *string2 = "123DC8";
   int length;

   length = strspn(string1, string2);
   printf("strings different at position
                        %d\n",length); // 3
   return 0;
}
```

strcspn()

Doorzoekt een string.

Prototype:

```
size_t strcspn(const char *s1, const char *s2);
```

Geeft de lengte van het initiële segment van `s1` terug dat volledig bestaat uit tekens niet in `s2`.

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>

int main()
{
   char *string1 = "1234567890";
   char *string2 = "747DC8";
   int length;

   length = strcspn(string1, string2);
   printf("strings intersect at position
                        %d\n", length); // 3

   return 0;
}
```

strpbrk()

Doorzoekt een string voor de eerste plaats waar een willekeurig teken uit de tweede string voorkomt.

Prototype:

```
char *strpbrk(const char *s1, const char *s2);
```

Geeft een pointer terug naar de eerste plaats waar een van de tekens uit `s2` in `s1` voorkomt. Indien geen van de `s2` tekens in `s1` voorkomen, wordt `NULL` teruggegeven.

```
#include <stdio.h>
#include <string.h>

int main()
{
   char *string1 = "abcdefghijklmnopqrstuvwxyz";
   char *string2 = "onm";
   char *ptr;

   ptr = strpbrk(string1, string2);

   if (ptr)
   {
      printf("found first character:
                        %c\n",*ptr);// 'm'
   }
   else
   {
      printf("strpbrk didn't find
                        character in set\n");
   }
   return 0;
}
```

strstr()

Zoekt de eerste plaats waar een substring in een andere string voorkomt.

Prototype:

```
char *strstr(const char *s1, const char *s2);
```

Geeft een pointer terug naar het element in `s1` dat `s2` bevat (wijst naar `s2` in `s1`), of `NULL` indien `s2` niet voorkomt in `s1`.

```
#include <stdio.h>
#include <string.h>

int main()
{
   char *str1 = "Borland International";
   char *str2 = "nation", *ptr;

   ptr = strstr(str1, str2);
   printf("The substring is: %s\n",
                         ptr); // "national"
   return 0;
}
```

strlen()

Berekent de lengte van een string.

Prototype:

```
size_t strlen(const char *s);
```

Geeft het aantal tekens in `s` terug, de eindnul wordt niet meegeteld.

```
#include <stdio.h>
#include <string.h>

int main()
{
   char *string = "Linux";

   printf("%d\n", strlen(string)); // 5
   return 0;
}
```

strtok()

Zoekt in `s1` naar het eerste teken dat niet voorkomt in in `s2`.

Prototype:

```
char *strtok(char *s1, const char *s2);
```

`s2` definiëert scheidingstekens. `strtok()` interpreteert de string `s1` als een reeks tokens gescheiden door een reeks tekens uit `s2`.

Indien geen tokens gevonden worden in `s1`, wordt `NULL` teruggegeven.

Indien het token gevonden is, wordt een nulteken in `s1` geschreven volgend op het token, en `strtok` geeft een pointer terug naar het token.

Volgende oproepen van `strtok()` met `NULL` als eerste argument gebruiken de vorige `s1` string, vanaf het laatst gevonden token.

```
#include <stdio.h>
#include <string.h>

int main()
{
  char s[] ="aze ry   iio sdf";
  char *p;

  p = strtok(s," ");
  while(p != NULL)
```

```
  {
     printf("%s\n",p);
     p = strtok(NULL," ");
  }

  return 0;
}
```

Het bovenstaande voorbeeld splitst een string op in woorden. De spatie is het scheidingsteken.

Structuren

`malloc()` en `free()`

Met de functies `malloc()` en `free()` kan je blokken geheugen op dynamische wijze reserveren en vrijgeven. Met `malloc()` doe je de reservatie en met `free()` wordt een blok geheugen vrijgegeven. De prototypes van beide functies zien er zo uit:

```
void *malloc(size_t size);
void free(void *ptr);
```

De functie `malloc()` verwacht als parameter de grootte van het blok geheugen in bytes. Als resultaat geeft de functie een pointer naar de eerste byte van het blok terug. Een blok geheugen dat je zo gereserveerd hebt, moet je met `free` teruggeven. Als je dat niet doet (vrijwillig of onvrijwillig), is dat een fout. Er is dan sprake van een *geheugenlek*. Met het commando `valgrind` in Linux kan je nagaan of er geheugenlekken in een programma voorkomen.

Het volgende voorbeeld toont hoe je dynamisch een array van 100 `int`'s kan reserveren. Je moet wel de grootte van het blok geheugen berekenen en aan `malloc()` doorgeven. In dit geval is dit `100 * sizeof(100)`. Je ziet in het voorbeeld dat het adres dat `malloc()` teruggeeft, met een cast omgezet wordt van `void *` naar `int *`.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *p = (int *) malloc(100 * sizeof(4));

  for (int i=0; i<100; i++)
  {
     p[i] = i*i;
  }

  free(p);
  return 0;
}
```

Het voordeel van de werkwijze in dit voorbeeld is de mogelijkheid om de grootte van het blok geheugen pas te bepalen op het moment dat de reservatie gebeurt. Hierdoor kan een programma zich aanpassen aan de geheugenbehoefte van het moment zelf. Als je arrays met een vaste afmeting declareert, heb je die flexibiliteit niet. Merk op dat de handelswijze in het voorbeeld overeenkomt met de wijze waarop arrays in Java worden gereserveerd. In Java is dat altijd dynamisch.

Een verschil tussen Java en C/C++ is het feit dat in Java de vrijgave van blokken geheugen automatisch verloopt. Het dynamisch geheugen wordt bijgehouden in een zone die men de *heap* noemt. Dit is geldig voor zowel C/C++ als Java. In C moet je na gebruik het geheugen vrijgeven met `free()`; in C++ doe je dat met de `delete` operator. In Java mag je deze stap achterwege laten. In Java worden regelmatig alle blokken geheugen opgespoord die niet meer in gebruik zijn. Deze blokken worden dan automatisch vrijgegeven. Deze bewerking loopt in de achtergrond en noemt men *garbage collection*. Je moet dus begrijpen dat de correct werkende C/C++ programma's geen geheugenlekken mogen hebben. Bij langdurig

lopende programma's kunnen geheugenlekken een tekort aan dynamisch geheugen veroorzaken. Dit leidt tot `malloc()` oproepen die een `NULL` als resultaat teruggeven om te melden dat er geen geheugen meer beschikbaar is. Dit kan pointerfouten veroorzaken en een crash van het lopende programma als gevolg.

Types maken met `typedef`

We bespreken eerst de mogelijkheid om aan zelf gedefiniëerde types een naam te geven. Dit bespaart schrijfwerk bij het declareren van variabelen en parameters. Dit laat ook toe om globaal het type van een soort gegeven te wijzigen. We creëren types `METER`, `VECTOR` en `STRING` met behulp van `typedef`.

```
typedef int METER;
typedef double VECTOR[3];
typedef char STRING[80];
```

Met `typedef` leggen we vast dat het nieuwe type `METER` overeenkomt met het type `int`. Net zoals bij `#define` wordt het type `METER` in hoofdletters geschreven. Dit is geen verplichting, maar wordt door veel programmeurs toegepast om het onderscheid te maken tussen constanten, types en variabelen. Met deze nieuwe types declareren we variabelen.

```
METER afstand;
VECTOR vect1,vect2;
STRING tekst="abcdefghijklmnopqrstuvwxyz";
```

Een zelf gedefiniëerd type kan ook bij functies gebruikt worden.

```
void druk(STRING v)
{
   printf("dit is de string: %s\n", v);
}
```

Het is zo dat het gebruik van `typedef` de leesbaarheid van het programma verbetert.

Structuur[22] [23]

Met een structuur kunnen we een type ontwerpen, dat gegevens van een verschillende soort samenbrengt. We doen dit met het woord `struct`.

```
struct boek
{
   char titel[40];
   char auteur[40];
   float prijs;
};
```

Deze declaratie creëert een structuur met de naam boek. Ze bestaat uit 2 `char` arrays en een `float`. Met deze structuur declareren we een variabele.

```
struct boek roman1;
```

Een variabele kan gedeclareerd worden met initialisatie.

```
struct boek roman2 =
{
   "De loteling","Conscience",17.5
};
```

De `roman2` bevat de velden `titel`, `auteur` en `prijs`. Het veld `titel` krijgt de waarde `"De loteling"`, auteur wordt `"Conscience"` en prijs wordt `17.5` .

We kunnen een structuur vastleggen als een nieuw type met `typedef`.

[22] Vanaf hier mag je het hoofdstuk overslaan als je van plan bent om het C++ deel ook te leren. In C++ kan jet met `class` werken ter vervanging van `struct`.

[23] Sommige studenten denken dat in C objectoriëntatie mogelijk is omdat er `struct`'s bestaan. Dit klopt niet, in `struct`'s zijn geen methoden toegelaten, in de C++ `class` is dat wel het geval.

```
typedef struct
{
   double x;
   double y;
} PUNT;
```

Merk op dat er geen structuurnaam, maar alleen een typenaam is. Met het type `PUNT` declareren we enkele variabelen.

```
PUNT p1,p2,p3;
```

Deze variabelen hebben elk de velden `x` en `y`, waarin een `double` opgeslagen wordt. De toegang tot velden wordt getoond in de volgende functie.

```
PUNT helft(PUNT pa,PUNT pb)
{
   PUNT pc;

   pc.x = (pa.x + pb.x)/2;
   pc.y = (pa.y + pb.y)/2;
   return( pc );
}
```

Deze functie ontvangt 2 variabelen van het type `PUNT` en levert een resultaat van hetzelfde type. We kunnen een veld bereiken door de variabelenaam uit te breiden met een punt en de veldnaam.

```
p1.x = 1;
p1.y = 2;
p2 = p1;/* x en y velden worden gekopieerd */
p3 = helft( p1, p2 );
```

Wanneer een structuurvariabele wordt toegekend aan een andere, wordt de hele structuur gekopieerd. Dus elk veld van de ene variabele wordt gekopieerd naar elk veld van de andere. Dit is dus *call by value*.

Arrays van structuren

We declareren polygoon als een array van 50 elementen van het type `PUNT`.

```
PUNT polygoon[50];
```

Dit zijn de velden van het eerste element van `polygoon`.

```
polygoon[0].x
polygoon[0].y
```

Dit zijn de velden van het laatste element.

```
polygoon[49].x
polygoon[49].y
```

Het volgende voorbeeld berekent de lengte van een polygoon als de som van de afstand tussen de opeenvolgende punten.

```
double afstand(PUNT pa, PUNT pb)
{
   double x,y;

   x = pa.x - pb.x;
   y = pa.y - pb.y;
   return( sqrt(x*x + y*y) );
}

lengte = 0;
for (int i=0; i<49; i++)
{
```

```
   lengte += afstand(polygoon[i], polygoon[i+1]);
}
```

Bij de oproep van `afstand()` zien we de notatie `polygoon[i]`. Deze uitdrukking is van het type `PUNT` en dit komt overeen met de declaratie van de formele parameters van `afstand()`.

Pointers naar structuren

Net zoals een pointer naar een `int` type kunnen we een pointer declareren die naar het type `PUNT` wijst.

```
PUNT *p_ptr;
```

We zien dat door het gebruik van het type `PUNT` in plaats van de hele structuurnotatie, de declaratie leesbaar blijft. Vermits `p_ptr` een pointervariabele is, kan hierin het adres van een `PUNT` variabele geplaatst worden.

```
p_ptr = &p1;
```

Om een veld van de aangewezen structuur te bereiken, schrijven we:

```
(*p_ptr).x
```

Deze waarde is dezelfde als `p1.x` omdat `p_ptr` naar `p1` wijst. De haken zijn nodig omdat `.` een hogere prioriteit heeft dan `*` . We schrijven dit in een andere vorm.

```
p_ptr->x
```

De notatie `->` is dus een samentrekking van het sterretje en het punt.

In de functie `maaknul()` wordt geen structuur doorgegeven maar wel een pointer naar een structuur. Dit is nodig omdat de `PUNT` variabele waarvan het adres doorgegeven wordt aan de functie, gewijzigd wordt.

```
void maaknul(PUNT *p)
{
   p->x = 0;
   p->y = 0;
}
```

Dit is het gebruik van de functie:

```
PUNT p1,p2,p3;

maaknul( &p1 );
maaknul( &p2 );
maaknul( &p3 );
```

Structuur binnen structuur

Het is mogelijk om als type voor een veld een zelfgedefiniëerd type te gebruiken.

```
typedef struct
{
   int jaar;
   int maand;
   int dag;
} DATUM;

typdef struct
{
   DATUM van;
   DATUM tot;
} PERIODE;

PERIODE contract;
```

De variabele `contract` is van het type `PERIODE` en bestaat dus uit de velden `van` en `tot`. Deze velden zijn op hun beurt structuren. Ze bestaan uit de velden `jaar`, `dag` en `maand`.

De velden kunnen zo bereikt worden:

```
contract.tot.jaar
contract.van.dag
```

Deze uitdrukkingen moeten we zo interpreteren:

```
(contract.van).dag
```

De `.` operator groepeert dus van links naar rechts.

Unions

Soms is het nodig om een bepaalde waarde onder verschillende vormen bereikbaar te maken. Dit doen we met `union`.

```
typedef union
{
   float fwaarde;
   long lwaarde;
} MASKER;
```

De schrijfwijze is identiek met die van `struct`. We hoeven maar het woord `struct` te vervangen door `union`. De betekenis is anders. In tegenstelling tot `struct` wordt hier maar één keer geheugenruimte gereserveerd. In dit voorbeeld hebben de types `float` en `long` dezelfde afmeting. Er wordt dus 4 bytes geheugen gereserveerd. Indien de velden een verschillende afmeting hebben, dan bepaalt het grootste veld de hoeveelheid gereserveerd geheugen.

```
MASKER getal;
getal.fwaarde = 3.14159;
printf("voorstelling van %f in hex is %lx\n",
getal.fwaarde,getal.lwaarde);
```

Resultaat:

```
voorstelling van 3.141590 in hex is 40490fcf
```

In dit voorbeeld wordt er voor `getal` 4 bytes gereserveerd. Dit geheugen is bereikbaar met twee namen: `getal.fwaarde` en `getal.lwaarde`. We plaatsen een float-constante in `getal` en daarna toont `printf()` op welke wijze dit opgeslagen wordt.

Bitvelden

In sommige gevallen is het bereik van de werkelijke waarden van een veld slechts een fractie van het maximale bereik. In dat geval is het wenselijk om de velden te declareren op bitniveau.

```
typedef struct
{
   unsigned int jaar:12;/* 0 - 4095*/
   unsigned int maand:4;/* 0 - 15*/
   unsigned int dag:5;/* 0 - 31*/
   unsigned int ongeveer:1;/* 0 - 1*/
} CDATUM;
```

Elk veld heeft nu een aangepast bereik. Dit verkrijgen we door na elke veldnaam dubbele punt en bitbreedte bij te voegen. De veldbreedte in bit mag niet groter zijn dan de bitbreedte van het gebruikte type. Als type voor een bitveld mogen we alleen maar gehele types gebruiken.

De veldnaam mag weggelaten worden. Hiermee kan men ongebruikte bits overslaan.

```
struct metbits
{
   int i:2;
```

```
   unsigned j:5;
   int:4;
   int k:1;
   unsigned m:4;
} a,b,c;
```

De bitverdeling van `a`, `b` en `c` ziet er als volgt uit:

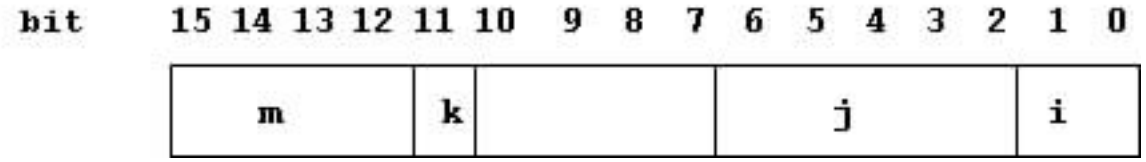

Figure 8: Bits in een struct

De bits 7 tot 10 zijn niet gebruikt.

Structuren en lijsten

Dit deel is er enkel om historische redenen. Vroeger was de gelinkte lijst (en aanverwanten) de enige mogelijkheid om dynamische datastructuren te maken. Nu is er STL in C++ waar deze technieken ingekapseld zijn.

Zelfreferentiële structuren

Dit zijn structuren die één of meerdere pointers bevatten die naar eenzelfde soort structuur verwijzen. Deze structuren kunnen gebruikt worden om gegevens op een dynamische manier te organiseren. Men maakt bijvoorbeeld een ketting van zelfreferentiële structuren. Elk knooppunt in deze ketting bevat 1 of meerdere gegevenselementen en bevat ook een verwijzing naar het volgende knooppunt.

Hier is een voorbeeld van een zelfreferentiële structuur.

```
struct knoop
{
   int  data;
   struct knoop *verder;
};
typedef struct knoop KNOOP;
```

Het veld `verder` in deze structuur verwijst naar een andere variabele van het type `struct knoop`. Het type `KNOOP` is een synoniem voor `struct knoop`. We declareren enkele variabelen.

```
KNOOP a,b,c;
```

Deze variabelen worden gevuld met gegevens.

```
a.data = 1;
b.data = 2;
c.data = 3;
a.verder = b.verder = c.verder = NULL;
```

De velden verder worden voorlopig niet gebruikt en ze krijgen daarom de waarde `NULL` (is gelijk aan 0). `NULL` wordt gebruikt om aan te geven dat een pointer naar niets wijst. De huidige toestand stellen we grafisch voor.

We maken nu de verbinding tussen `a`, `b` en `c`.

```
a.verder = &b;
b.verder = &c;
```

Nu zijn de gegevens vanuit `a` bereikbaar.

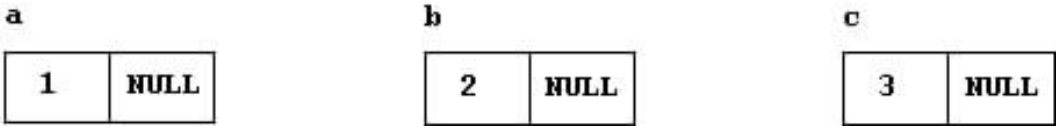

Figure 9: Drie structs zonder koppeling

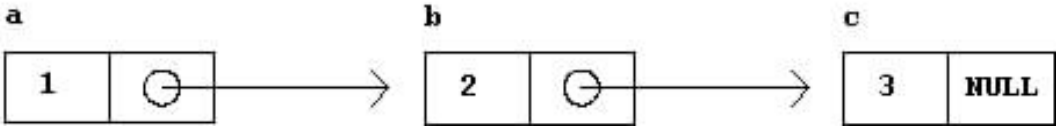

Figure 10: Drie gekoppelde structs

```
a.data-->1
a.verder->data2
a.verder->verder->data3
```

Niet gesorteerde gebonden lijsten

De gegevensorganisatie die we daarnet besproken hebben, is een gebonden lijst. We hebben het nu verder over functies die een niet-gesorteerde gebonden lijst manipuleren. We creëren een pointervariabele die naar het eerste element wijst.

```
KNOOP *p_eerste = NULL;
```

Deze variabele wordt met `NULL` geïnitialiseerd. Hiermee wordt aangegeven dat de lijst leeg is.

```
void voegbij(KNOOP **ptr, int getal)
{
   register KNOOP *nieuw_p;

   nieuw_p = (KNOOP *) malloc(sizeof(KNOOP));
   nieuw_p->data = getal;
   nieuw_p->verder = *ptr;
   *ptr = nieuw_p;
}
```

Deze functie kan zo gebruikt worden:

```
voegbij( &p_eerste, 4 );
voegbij( &p_eerste, 5 );
voegbij( &p_eerste, 6 );
```

Het eerste wat opvalt, is de formele parameter `KNOOP **ptr`. Dit is een dubbele pointer[24]: `ptr` heeft als inhoud het adres van een pointer die wijst naar een `KNOOP`. De actuele parameter is van hetzelfde type: `&p_eerste` is het adres van een pointer. We geven niet de inhoud van een pointer door, maar wel het adres van die pointer. Dit is nodig omdat `p_eerste` in de functie gewijzigd moet kunnen worden. Dit gebeurt als het eerste element van de lijst gewist wordt of als er een ander element het eerste wordt.

Het eerste wat `voegbij()` doet, is het oproepen van `malloc()`. Dit is een functie die geheugen reserveert. De functie verwacht als actuele parameter het aantal benodigde bytes en levert als resultaat een pointer naar het aangevraagde geheugen. In dit geval hebben we geheugen nodig voor een element van het type `KNOOP`: dit is `sizeof(KNOOP)` bytes. Het resultaat van `malloc()` is een pointer naar `char`. Dit adres wordt met een cast omgezet tot een pointer naar `KNOOP`. Het bij te voegen getal wordt in het veld `data` geplaatst. Het veld `verder` van het nieuwe element moet nu wijzen naar het eerste element van de oude lijst.

[24] Dit soort notaties kan je anno 2016-2017 beter vermijden.

```
nieuw_p->verder = *ptr;
```

Het nieuwe element wijst dus naar het element dat vroeger door `p_eerste` aangewezen werd. `ptr` bevat het adres van `p_eerste`. Dus `*ptr` is hetzelfde als de inhoud van `p_eerste`. Tot slot wordt `p_eerste` gewijzigd.

Dit gebeurt onrechtstreeks:

```
*ptr = nieuw_p;
```

`p_eerste` wijst naar het nieuwe element en dit op zijn beurt wijst naar de oude lijst.

Toestand voor het bijvoegen van het getal 5:

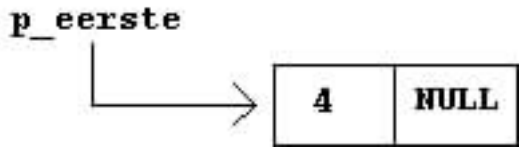

Figure 11: Lijst met 1 element

Toestand erna:

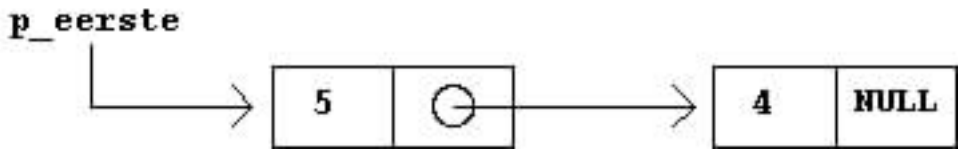

Figure 12: Lijst na het bijvoegen van een element

De inhoud van een lijst kan zichtbaar gemaakt worden met de functie `druk()`.

```
void druk(KNOOP *ptr)
{
   while (ptr != NULL)
   {
      printf("%d\n", ptr->data);
      ptr = ptr->verder;
   }
}
```

De functie wordt zo gebruikt:

```
druk(p_eerste);
```

De formele parameter `ptr` is een kopie van de inhoud van `p_eerste`. Deze kopie mag zonder meer gewijzigd worden zonder dat `p_eerste` verandert.

De pointer doorloopt de hele lijst tot het einde bereikt is en drukt bij elke herhaling een getal op het scherm.

Hier is een andere versie:

```
void druk(KNOOP *ptr)
{
   if (ptr != NULL)
   {
      printf("%d\n", ptr->data);
```

```
      druk( ptr->verder );
   }
}
```

Deze versie werkt recursief. Dit wil zeggen dat `druk()` zichzelf oproept. De functie drukt het getal, dat aangeduid wordt door `ptr` en drukt dan de rest van de lijst door zichzelf nog eens op te roepen.

De volgende functie zoekt een getal in een lijst en veegt het uit als het gevonden wordt.

```
void veeguit(KNOOP **ptr; int getal)
{
   KNOOP **p1, *p2;

   // zoeken
   p1 = ptr;
   while ( *p1 != NULL && (*p1)->data != getal)
   {
      p1 = &((*p1)->verder);
   }

   if (*p1 != NULL) // gevonden
   {
      // uitvegen
      p2 = *p1; // bewaar adres gevonden element
      *p1 = (*p1)->verder;
      free(p2); // geheugenvrijgave
   }
   else
      printf("niet gevonden\n")
}
```

Aan deze functie wordt het adres van `p_eerste` doorgegeven, omdat ook hier `p_eerste` gewijzigd moet kunnen worden. Na het starten wordt een kopie gemaakt naar `ptr`. Deze pointer wordt gebruikt om telkens op te schuiven naar het volgende element tijdens het zoeken. Dit proces gaat verder zolang het einde van de lijst niet bereikt is

```
*p1 != NULL
```

en het getal niet gevonden is.

```
(*p1)->data != getal
```

We zien telkens een `*` voor `p1`. Dit is nodig omdat `p1` het adres bevat van een `KNOOP` pointer. `*p1` is dus een pointer naar `KNOOP`.

Tijdens de herhaling wordt `p1` gewijzigd: `p1` wijst dan naar het adres van de pointer die wijst naar het volgende element.

Welke waarde krijgt `p1`?

```
*p1                 // is adres huidige KNOOP element
(*p1)->verder       //    adres volgende KNOOP element
&((p1)->verder)     //    adres van de pointer die het adres bevat
                    //    van het volgende KNOOP element
```

Als de herhaling stopt en `*p1` is `NULL`, dan is het einde van de lijst bereikt en is het getal niet gevonden. In het andere geval moet het gevonden element geschrapt worden.

Toestand voor het uitvegen van 5:

Dit is het uitvegen:

```
veeguit( &p_eerste, 5);
```

Toestand erna:

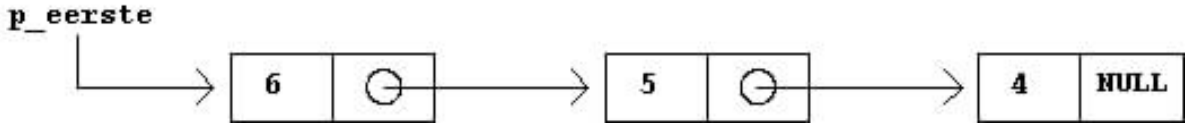

Figure 13: Lijst voor het uitvegen

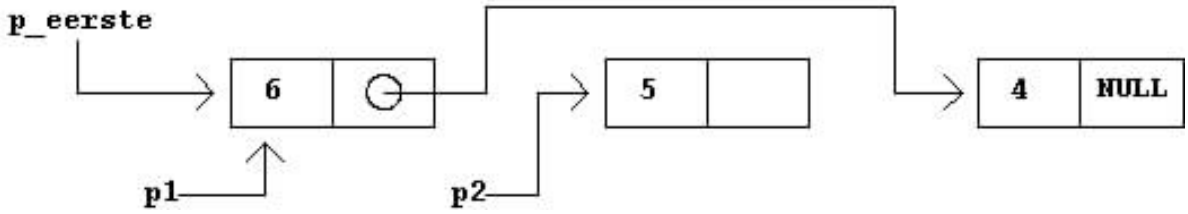

Figure 14: Lijst na het uitvegen

Na het vinden van het getal 5 wijst `*p1` naar het gevonden `KNOOP` element. Dit adres wordt opzijgezet in `p2` en de pointer `*p1` krijgt een nieuwe waarde. Hierdoor wordt het uit te vegen element overgeslagen.

Merk op dat er zich een speciale situatie voordoet als het eerste element van de lijst geschrapt wordt. In dat geval doet de `while` opdracht geen herhaling en bevat `p1` het adres van `p_eerste`. Hieruit volgt dat `p_eerste` gewijzigd wordt.

Nogmaals moet herhaald worden dat dit type constructies in C kan vermeden worden door over te schakelen naar C++ en daar de STL `vector` of andere te gebruiken.

Bestandsin- en uitvoer

In dit hoofdstuk worden een aantal functies beschreven voor het lezen en schrijven van bestanden. Tenzij anders aangegeven zijn de prototypes allemaal terug te vinden in stdio.h. De in- en uitvoerfuncties maken deel uit van de C standaard en maken gebruik van filepointers. Meestal bestaan er op elk systeem ook nog functies die dichter bij de hardware staan en specifiek zijn voor het desbetreffende operating system. Om die reden worden deze functies hier niet beschreven.

`fopen()`

Vooraleer er gelezen of geschreven wordt van of naar een bestand, moet een bestand geopend worden. Bij deze actie wordt een filepointer geassocieerd met de file. In de overige functies moet een filepointer meegegeven worden als referentie naar het bestand. In C zijn er twee voorgedefiniëerde filepointers voor in- en uitvoer: `stdin`, `stdout` en `stderr`. Ze worden gebruikt voor de invoer van het toetsenbord, de uitvoer naar het scherm en foutuitvoer.

Prototype:

```
FILE *fopen(const char *filename, const char *mode);
```

Hierbij is `filename` een string die het pad van het bestand bevat en kunnen in de string `mode` de volgende letters voorkomen.

Letter	Betekenis
r	open enkel om te lezen
w	creëer voor schrijven; overschrijft een bestaand bestand
a	bijvoegen, open voor schrijven op het einde van het bestand of creëer voor schrijven
+	nog een letter volgt (bv combinatie lezen en schrijven)
b	open in binaire modus

t	open in tekstmodus

In de binaire modus worden bytes letterlijk geschreven en gelezen. In de tekstmodus wordt de carriage return/linefeed combinatie vervangen door een enkele linefeed. Bij het lezen van een tekstbestand wordt dus nooit een carriage return aan het programma gegeven. Bij het schrijven gebeurt het omgekeerde.

De functie `fopen()` geeft als resultaat een filepointer of `NULL` bij fout.

`fclose()`

Met deze functie wordt een bestand gesloten.

Prototype:

```
int fclose(FILE *stream);
```

De functie geeft als resultaat een 0 bij succes of `EOF` bij fout.

```
// Programma dat een kopie maakt van een bestand

#include <stdio.h>

int main()
{
  FILE *in, *out;

  if ((in = fopen("file.dat",
                  "rt")) == NULL)
  {
     fprintf(stderr, "Cannot open input file.\n");
     return 1;
  }

  if ((out = fopen("file.bak",
                  "wt")) == NULL)
  {
     fprintf(stderr, "Cannot open output file.\n");
     return 1;
  }

  while (!feof(in))
  {
     fputc(fgetc(in), out);
  }

  fclose(in);
  fclose(out);
  return 0;
}
```

`ferror()`

Deze macro geeft als resultaat een waarde verschillend van nul als er een fout is opgetreden bij deze filepointer.

Prototype:

```
int ferror(FILE *stream);
```

perror()

Deze functie drukt een foutmelding op het scherm via `stderr`.

Prototype:

```
void perror(const char *s);
```

Eerst wordt de string `s` gedrukt. Daarna volgt een dubbele punt en een foutmelding die overeenkomt met de huidige waarde van `errno`.

strerror()

Deze functie geeft als resultaat een foutmelding-string die overeenkomt met het doorgegeven foutnummer.

Prototype (ook in `string.h`):

```
char *strerror(int errnum);
```

_strerror()

Deze functie geeft een string met een foutmelding terug. Het formaat is zoals bij `perrror`.

Prototype (ook in `string.h`):

```
char *_strerror(const char *s);
```

Het volgende voorbeeld toont hoe foutmeldingen op het scherm kunnen geplaatst worden.

```
#include <stdio.h>
#include <errno.h>

int main()
{
  FILE *stream;

  // open a file for writing
  stream = fopen("DUMM.FIL", "r");

  // force an error condition by attempting to read
  (void) getc(stream);

  if (ferror(stream)) // test for an error on the stream
  {
     // display an error message
     printf("Error reading from DUMMY.FIL\n");
     perror("fout");

     printf("%s\n",strerror(errno));

     printf("%s\n", _strerror("Custom"));

     // reset the error and EOF indicators
     clearerr(stream);
  }

  fclose(stream);
  return 0;
}
```

fwrite()

Met deze functie kan informatie naar een bestand geschreven worden. De functie schrijft `n` elementen van afmeting `size` bytes naar het bestand. De pointer `ptr` wijst naar de te schrijven informatie.

Prototype:

```
size_t fwrite(const void *ptr, size_t size, size_t n,
                    FILE *stream);
```

De functie geeft als resultaat het aantal elementen (niet bytes) dat weggeschreven is.

Het volgende voorbeeld toont hoe een structuur naar een binair bestand geschreven wordt.

```
#include <stdio.h>

struct mystruct
{
   int i;
   char ch;
};

int main()
{
  FILE *stream;
  struct mystruct s;

  // open file
  if ((stream = fopen("TEST.$$$",
                    "wb")) == NULL)
  {
     fprintf(stderr, "Cannot open output file.\n");
     return 1;
  }
  s.i = 0;
  s.ch = 'A';

      /* write struct s to file */
  fwrite(&s, sizeof(s), 1, stream);
  fclose(stream); /* close file */
  return 0;
}
```

Deze werkwijze is te vergelijken met de Java objectserialisatie. Dat levert ook een binair bestand op.

`fread()`

Deze functie leest uit een bestand. Er worden `n` elementen van afmeting `size` bytes in de array `ptr` gelezen.

Prototype:

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

De functie levert als resultaat het aantal elementen (niet bytes) dat effectief gelezen is.

`fseek()`

Deze functie verplaatst de wijzer die de positie aangeeft waar eerstvolgend gelezen of geschreven wordt.

Prototype:

```
int fseek(FILE *stream, long offset, int fromwhere);
```

`offset` is de nieuwe positie relatief ten opzichte van de positie gespecificeerd met `fromwhere`.

De functie geeft 0 bij succes of nonzero bij fout.

De parameter `fromwhere` kan een van de volgende waarden zijn:

- `SEEK_SET` verplaats vanaf het begin van het bestand

- `SEEK_CUR` verplaats vanaf de huidige positie
- `SEEK_END` verplaats vanaf het einde van het bestand

fgets()

Deze functie leest een regel uit een bestand.

Prototype:

```
char *fgets(char *s, int n, FILE *stream);
```

De parameter `n` geeft aan voor hoeveel tekens er plaats is in de buffer `s`.

Bij succes, wordt de string `s` of `NULL` bij einde van het bestand of fout teruggegeven.

```
#include <string.h>
#include <stdio.h>

int main()
{
  FILE *stream;
  char string[] = "This is a test";
  char msg[20];

  // open a file for update
  stream = fopen("DUMMY.FIL", "w+");

  // write a string into the file
  fwrite(string, strlen(string), 1, stream);

  // seek to the start of the file
  fseek(stream, 0, SEEK_SET);

  // read a string from the file
  fgets(msg, strlen(string)+1, stream);

  // display the string
  printf("%s", msg);

  fclose(stream);
  return 0;
}
```

fputs()

Met deze functie wordt een regel naar een bestand geschreven.

Prototype:

```
int fputs(const char *s, FILE *stream);
```

De functie geeft als resultaat bij succes het laatst weggeschreven teken of `EOF` bij fout.

fgetc()

Met deze functie wordt een teken gelezen uit een bestand.

Prototype:

```
int fgetc(FILE *stream);
```

De functie geeft het teken of `EOF` terug.

`fputc()`

Met deze functie wordt een teken naar een bestand geschreven.

Prototype:

```
int fputc(int c, FILE *stream);
```

`fprintf()`

Dit is de file-variant van `printf`.

Prototype:

```
int fprintf(FILE *stream, const char *format, ...);
```

De functie geeft als resultaat het aantal geschreven bytes of `EOF` bij fout.

`fscanf()`

Dit is de file-variant van `scanf`.

Prototype:

```
int fscanf(FILE *stream, const char *format, ... );
```

De functie geeft als resultaat het aantal in variabelen opgeborgen waarde.

Gereserveerde woorden in C99

```
auto      enum     restrict unsigned
break     extern   return   void
case      float    short    volatile
char      for      signed   while
const     goto     sizeof   _Bool
continue  if       static   _Complex
default   inline   struct   _Imaginary
do        int      switch
double    long     typedef
else      register union
```

Prioriteiten van de operatoren

Operator	Groepering
parse_error	
parse_error	rechts -> links (unair)
parse_error	links -> rechts
parse_error	links -> rechts
parse_error	
parse_error	links -> rechts
parse_error	links -> rechts
parse_error	links -> rechts
parse_error	links -> rechts
parse_error	links -> rechts
parse_error	links -> rechts
parse_error	links -> rechts
parse_error	links -> rechts
parse_error	links -> rechts
parse_error	rechts -> links
parse_error	links -> rechts

Werking stackframe in C

In deze appendix wordt uitgelegd hoe de C compiler code genereert die stackframes mogelijk maken. Stackframes zijn het mechanisme dat mogelijk maakt om geheugenruimte te reserveren voor de lokale variabelen. Een programmeertaal die dit mechanisme ondersteunt, laat meestal recursie toe. Zonder stackframes is recursie niet mogelijk. Als we bijvoorbeeld het volgende voorbeeld bekijken, dan zie meteen de recursie. De functie `reken()` roept zelf op.

```
int reken(int n)
{
   int som = 0;

   if (n > 0)
   {
      som = n + reken(n - 1);
   }
   return som;
}
```

Als we `reken(6)` starten, is het resultaat 21. Er zijn in totaal 7 oproepen: `reken(6)`, `reken(5)`, `reken(4)`, `reken(3)`, `reken(2)`, `reken(1)` en `reken(0)`. Bij elke oproep ontstaat er een nieuwe variabele `som`. Vermits het telkens andere variabelen zijn, hebben die ook allemaal een ander adres. Dat is de reden waarom de ruimte van deze variabelen op de stack wordt gereserveerd. Schematisch ziet er dat dan zo uit:

Stack met meerdere lokale variabelen

Figure 15: Stack met meerdere lokale variabelen

De stack is een geheugengebied dat meestal bovenaan in het geheugen zit en dat naar beneden toe kan groeien. De stackpointer, `SP` in de figuur, wijst naar de onderzijde van de stack. Als er geheugenruimte op de stack wordt gereserveerd, volstaat het om de stackpointer te verlagen. De reservatie kan ongedaan worden gemaakt door de stackpointer te verhogen.

We gebruiken het onderstaande C voorbeeld om te demonstreren hoe de compiler de nodige assembler instructies genereert. Het programma heeft een globale variabele `c`, een functie `som` met parameters `a` en `b`. De functie heeft ook twee lokale variabelen `d` en `e`. Alle variabelen behalve `c` krijgen hun plaats op de stack. Hoe dat precies gebeurt, kunnen we pas ontdekken als we de assemblercode analyseren.

```
int c;

int som(int a, int b)
{
   int d = 50;
   int e = 60;

   return a + b;
}

void main()
{
   c = som(200000, 300000);
}
```

Eerst moet de C broncode gecompileerd worden. We make hier gebruik van een cross-compiler [25] die code genereert voor de Motorola M68000 [26]. Deze processor heeft een relatief eenvoudige instructieset die bijgevolg goed geschikt is om de werking van stackframes uit te leggen.

```
/opt/m68k/bin/m68k-elf-gcc -S -m68000 -O0 test2.c
```

[25] Een *cross-compiler* is een compiler die code genereert voor een andere processor dan degene waarop de compiler zelf draait.

[26] De M68000 processor is ontstaan in de jaren 70-80. De instructieset is afgeleid van die van de PDP11. En de PDP11 is de computer waarvoor het eerst C compilers zijn ontwikkeld.

De optie `-S` zorgt ervoor dat de compilatie stopt na het genereren van assemblercode. Deze code zal je dan terugvinden in `test2.s`. Het lezen van dit bestand zonder enige zal wel lastig zijn, assembler behoort niet meer tot de onderwezen topics.

We zullen stap voor stap de instructies uitleggen. Eerst wordt snel de 68000 architectuur uitgelegd. De 68000 heeft 16 interne registers van elk 32 bit breed. `D0` tot `D7` zijn dataregisters en `A0` tot `A7` zijn adresregisters. Deze laatste kunnen gebruikt worden om adressen bij te houden. Het laatste adresregister A7 doet dienst als stackpointer. Je kan het ook bereiken met de naam `SP`. Verder is er uiteraard ook een programcounter `PC`. Dit register geeft aan welke instructie als volgende uitgevoerd zal worden. De `PC` bevat dus het adres van de volgende instructie.

```
#NO_APP
    .file   "test2.c"
    .comm   c,4,2
    .text
    .align  2
    .globl  som
    .type   som, @function
som:
    link.w %fp,#-8
    moveq #50,%d0
    move.l %d0,-4(%fp)
    moveq #60,%d0
    move.l %d0,-8(%fp)
    move.l 8(%fp),%d0
    add.l 12(%fp),%d0
    unlk %fp
    rts
    .size   som, .-som
    .align  2
    .globl  main
    .type   main, @function
main:
    link.w %fp,#0
    move.l #300000,-(%sp)
    move.l #200000,-(%sp)
    jsr som
    addq.l #8,%sp
    move.l %d0,c
    nop
    unlk %fp
    rts
    .size   main, .-main
    .ident  "GCC: (GNU) 7.1.0"
```

We starten de uitleg vanaf de label `main:`. De `link` instructie slaan we even over. De volgende instructie is een `move`.

```
    move.l #300000,-(%sp)
```

Deze instructie neemt de constante `300000` en plaats die waarde op de stack. Het `#`-teken wijst op de *immediate* adressering. Dit betekent dat de betrokken waarde als constante in de instructie is opgeslagen. De constante staat hier als linker operand vermeld. De rechter operand is `-(%sp)`. De ronde haken wijzen op de indirecte adressering: de waarde gaat niet naar het `SP` register maar naar het geheugen dat door `SP` aangewezen wordt. Dit is zoals pointers in C. Het minteken geeft aan dat de stackpointer `SP` eerst verminderd wordt voordat de geheugenschrijfbewerking plaats vindt. En niet te vergeten, de `move` wordt als `move.l` geschreven. Dit betekent dat een 32 bit waarde naar het geheugen geschreven wordt.

De tweede `move.l` werkt gelijkaardig.

```
    move.l #200000,-(%sp)
```

Beide `move.l` instructies zorgen ervoor dat de twee parameters op de stack worden gepusht, eerst de tweede parameter en dan de eerste parameter. Door de parameters op de stack te plaatsen kan de functie `som` toegang krijgen tot deze waarden. De stack ziet er nu zo uit:

Stack na push parameters

Figure 16: Stack na push parameters

De volgende stap is de start van de functie `som()`. Hiervoor wordt de `jsr` (*J*ump to *S*ub*r*outine) instructie gebruikt.

```
jsr som
```

Deze instructie heeft eerst het terugkeeradres op de stack geplaatst en dan heeft `SP` het adres van de functie `som()` gekregen. De stack ziet er nu zo uit:

Stack na `jsr`

Figure 17: Stack na `jsr`

Nu gaan we verder in de functie `som()`. Hier is de eerste instructie. Dit is een `link` instructie.

```
link.w %fp,#-8
```

Deze instructie reserveert ruimte op de stack voor de lokale variabelen. In de instructie hierboven is dat 8 bytes. Dit is voor de variabelen `d` en `e`. De `link` instructie past ook de framepointer aan. Dat is het register dat dienst doet als pointer naar het gebied met de lokale variabelen. Meestal wordt daarvoor `A6` gebruikt en in de gegenereerde code wordt de naam `FP` gebruikt, dit is een synomiem voor `A6`. De taak van de framepointer is het vergemakkelijken van de toegang tot de lokale variabelen en de parameters. Er kan dan relatief ten opzichte van de framepointers geadresseerd worden.

De `link` instructie voert de volgende stappen uit.

- De huidige waarde van de framepointer `FP` wordt op de stack gepusht.
- De waarde van de stackpointer `SP` wordt naar de framepointer `FP`. Dit is het beginadres van het gebied met de huidige lokale variabelen.
- De stackpointer `SP` wordt verlaagd met hoeveelheid die met de instructie is meegegeven. Hier is dat 8 byte.

Vanaf dit moment kunnen de lokale variabelen gebruikt worden. En de stack ziet er nu zo uit:

Stack na `link`

Figure 18: Stack na `link`

Via de framepointer `FP` en een negatieve offset kunnen `d` en `e` hun waarde krijgen. De volgende vier instructies nemen dat voor hun rekening. Telkens wordt de constante in het register `D0` geladen en dan wordt de inhoud van `D0` naar het geheugen geschreven. De adressen zijn `-4(%fp)` en `-8(%fp)`.

```
moveq #50,%d0
move.l %d0,-4(%fp)
moveq #60,%d0
move.l %d0,-8(%fp)
```

Daarna wordt de som berekend. Hiervoor worden de waarden van de parameters opgehaald.

```
move.l 8(%fp),%d0
add.l 12(%fp),%d0
```

De eerste parameter wordt in `D0` geladen en de tweede parameter wordt hierbij opgeteld. De som blijft in `D0` staan. Het is via het register `D0` dat het resultaat van de functie wordt teruggegeven.

Tot slot wordt het stackframe verwijderd met de `unlnk` instructie. Hierdoor krijgt de framepointer zijn originale waarde terug.

```
unlk %fp
rts
```

Met de `rts` instructie wordt er teruggekeerd naar `main`. De eerste instructie die dan uitgevoerd wordt, is een correctie van de stackpointers. De parameters waren op de stack geplaatst en deze bewerkig moet ongedaan gemaakt worden. Dat gebeurt met de volgende instructie:

```
addq.l #8,%sp
```

Deze instructie telt 8 op bij `SP`, die dan dezelfde waarde heeft van voor de hele oproep van `som`.

Wanneer er meerdere geneste oproepen van functies uitgevoerd worden, ontstaat er een gelinkte lijst van alle stackframes. Deze begin van deze lijst wordt aangewezen door het `FP` register. Debuggers maken gebruik van dit mechanisme om alle stackframe met de bijbehorende lokale variabelen te tonen. Deze appendix heeft duidelijk gemaakt dat functieoproepen in C op een ingenieuze wijze gecompileerd worden zodat recursie en bijgevolg een lokale toestand voor elke oproep mogelijk is.

Bibliografie

- Borland 4585 Scotts Valley Drive, Scotts Valley, California 95066 USA
- Prentice Hall 1978 Brian W.Kernighan Dennis M.Ritchie
- AlKelley IraPohl The Benjamins/Cummings Publishing Company, Inc. 2725 Sand Hill Road, Menlo Park, California 94025 USA
- MitchellWaite StephenPrata DonaldMartin The Waite Group, Howard W. Sams & Company A Division of Macmillan, Inc., 4200 West 62nd Street, Indianapolis, Indiana 46268 USA
- A.Bellen J.Vandebroek KHLim Universitaire Campus, gebouw B, 3590 Diepenbeek
- Brian W.Kernighan Dennis M.Ritchie Academic Service Postbus 81, 2870 AB Schoonhoven

Deel C++

In deel II wordt de programmeertaal C++ behandeld. Vermits C++ een uitbreiding is van C, zijn beide delen opgenomen in deze cursustekst. Dit heeft als voordeel dat de student een volledig overzicht krijgt van beide talen en ook inzicht krijgt welke kenmerken tot C behoren en welke tot C++.

Zoals reeds eerder vermeld, gebeurt het in de praktijk regelmatig dat voor een bepaald probleem enkel een C compiler ter beschikking wordt gesteld en dat er daardoor enkel in C in een niet-objectgeoriënteerde stijl kan geprogrammeerd worden. Het kan ook gebeuren dat de compiler zowel C als C++ ondersteund; in dit geval is de keuze voor C++ te verkiezen. Daarom is het essentieel dat een student precies kan inschatten wanneer C en wanneer C++ inzetbaar is. Het opsplitsen van deze cursustekst in twee delen moet bijdragen tot deze competentie.

Dit deel behandelt alleen maar de de specifieke syntax en constructies van C++. Uiteraard komt hierdoor de objectoriëntatie aan bod. Het gebruik van STL en enkele vernieuwingen van C++11/C++14 staan in latere delen. Dit deel behandelt dus C++ volgens de standaarden van vóór C++11.

Inleiding

In de geschiedenis van de programmeertalen zijn er regelmatig nieuwe paradigma's opgedoken. Met elk van die paradigma's werd een nieuwe programmeerstijl verdedigd. Een voorbeeld hiervan is het *gestructureerd programmeren*, dat jaren geleden het tijdperk van de goto-loze programmeerstijl inluidde. Deze programmeerstijl was een stap vooruit in het schrijven van duidelijke en leesbare programma's. Een relatief recente programmeerstijl is het *objectgeoriënteerd* programmeren. Het is de verdienste van de programmeertaal C++ en later Java gevolgd door Python en Ruby om de objectgeoriënteerde programmeerstijl populair te maken. Een voorbeeld hiervan zijn de moderne 32 bit microcontrollers zoals de populaire www.embed.com. De complexiteit van de hardware in deze microcontrollers is hoger dan bij de 8 bit microcontrollers. Hierdoor is het lastig om die hardware vanuit C te bereiken. In C++ gaat dit beter door de hogere abstractie die objectoriëntatie toelaat.

Met dit deel krijg je opnieuw een kennismaking met een objectgeoriënteerde taal. De kennismaking verloopt aan de hand van C++ voorbeelden. Een andere mogelijkheid zou zijn gebruik maken van de recentere taal Java. Deze taal heeft veel weg van C++ maar is ontdaan van alle onhebbelijkheden die in C++ voorkomen. Java laat toe om een programma op verschillende platformen te draaien zonder te hercompileren. Java is ondertussen goed ingeburgerd als eerste OO programmeertaal in de academische bachelor. De belangrijkste reden om na Java toch nog C++ te onderwijzen is dat C++ nog veel gebruikt wordt en in bepaalde toepassingsdomeinen beter voldoet dan Java.

Eigenschappen van objectgeoriënteerde talen

Objectgeoriënteerde talen hebben een aantal specifieke kenmerken die ze onderscheiden van niet-objectgeoriënteerde talen. Deze eigenschappen zijn:

Inkapseling van gegevens en methoden

Het is een gekende techniek om gegevens die een sterke relatie met elkaar hebben, te groeperen in een structuur of wat met in objectgeoriënteerde termen een *klasse* noemt. Deze groepering maakt het gemakkelijker om het overzicht op de gegevens binnen een programma te bewaren.

Hiermee wordt bedoeld dat niet alleen gegevens maar ook acties die inwerken op deze gegevens worden gegroepeerd binnen een *klasse*. Deze techniek laat toe om de gegevens binnen een *klasse* af te schermen van de buitenwereld. Wie deze gegevens wil raadplegen of veranderen moet dit doen via speciale functies die bij de *klasse* horen. De toegang tot de gegevens is niet direct maar indirect. De acties of functies binnen een *klasse* worden in de objectgeoriënteerde wereld *methoden* genoemd. Het voordeel van deze techniek is de mogelijkheid om achteraf nog de wijze waarop de gegevens binnen de *klasse* opgeslagen worden, gemakkelijk te wijzigen.

Erfenis

Als op een zekere dag blijkt dat de gegevensopslag binnen een klasse uitgebreid moet worden, dan zal men in plaats van de klasse te wijzigen een afleiding maken van deze klasse. De nieuwe klassen erft alle gegevens en methoden van de klasse waarvan geërfd wordt. Aan de nieuwe klassen kunnen andere gegevens en methoden toegevoegd worden of kan een overgeërfde methode vervangen worden door een nieuwe versie. Dit is het principe van de erfenis.

Late of dynamische verbinding

In de klassieke programmeertalen zorgt de linker ervoor dat de werkelijke adressen van de functies ingevuld wordt bij elke CALL instructie. Dit is het principe van de vroege of statische verbinding. In objectgeoriënteerde talen kan deze verbinding uitgesteld worden tot het uitvoeren van het programma. Vlak voor de uitvoering van de CALL wordt het adres van de functies opgezocht. Dit mechanisme geeft een verhoogde flexibiliteit en is de kern bij het object georiënteerd programmeren.

In C++ kan je kiezen tussen vroege en late verbinding. In Java heb je die keuze niet meer; hier is het altijd de late verbinding.

Polymorfie

Deze term wordt gebruikt om aan te geven dat de oproep van een functie of methode soms tot een ander gedrag leidt. Het gedrag van de opgeroepen methode is afhankelijk van de gegevens. Deze speciale afhankelijkheid tussen gegevens en methode maakt het mogelijk om delen van programma's te ontwerpen die een zekere vorm van algemeenheid bewaren. Hiermee verhoogt de kans dat deze programmadelen later herbruikt kunnen worden. Het mechanisme om polymorfie toe te laten is de late verbinding. Een voorbeeld uit het dagelijkse leven maakt polymorfie duidelijk: als we iemand de opdracht geven om een voertuig te wassen dan zal de man (of vrouw) in kwestie zijn gedrag aanpassen naargelang hij (of zij) een gewone auto moet wassen of een autobus.

Eén van de objectgeoriënteerde talen is C++. Andere talen uit het verleden zijn Eiffel, Smalltalk, ObjectC, Simula en Turbo Pascal. Verder zijn er objectgeoriënteerde uitbreidingen op reeds bestaande talen; dit geldt onder andere voor Lisp en Prolog. Aan dit lijstje worden uiteraard de jongste talen Java, Javascript, Python en Ruby toegevoegd.

Geschiedenis van C++

De taal C++ is ontworpen door Bjarne Stroustrup en is volledig gebaseerd op C. C op zijn beurt is afgeleid van zijn voorganger BCPL. De commentaarstarter // die in C++ ingevoerd is, bestond al in BCPL. Heel wat concepten van C++ (de naam C++ werd in de zomer van 1983 bedacht) zijn afgeleid van andere programmeertalen. Het klasseconcept met afleidingen en virtuele functies werd van Simula67 overgenomen. De mogelijkheid om bewerkingstekens een andere betekenis te geven en om overal in het programma declaraties van variabelen te schrijven is van Algol68 overgenomen. Zo ook komen de ideeën voor templates en exceptions uit Ada, Clu en ML. C++ werd door de auteur ontworpen voor eigen gebruik. De eerste versie was geen eigen compiler maar wel een C naar C++ omzetter. Dit programma heette `cfront`. Door de stijgende populariteit van C++ bleek toch een standaardisatie noodzakelijk. Deze stap in gang gezet binnen ANSI als de X3J16 commissie in 1989. C++ werd een ISO standaard in 1998.

De evolutie van C++ loopt nog altijd verder. In 2011 is de toenmalige laatste standaard opgesteld. Deze staat bekend onder de informele naam *C++11*. Ondertussen bestaat ook C++14, is C++17 bijna goedgekeurd en is C++20 gepland.

Het belang van de komst van de recente standaarden C++11 en C++14 kan moeilijk onder schat worden. De meest in het oog springende innovatie is de invoering van de lambdafunctie. Deze voor C++ nieuwe techniek doet de taal verschuiven in de richting van de functionele programmeertalen. Alle recentere talen, en dus jonger dan C++, kennen de mogelijkheid om lambdafuncties te maken. Dit heeft voor een druk op C++ gezorgd zodat in de recente C++11 standaard de lambdafuncties zijn opgenomen.

Oorspronkelijk werd C ontworpen om assembler te vervangen. Ook C++ kan in deze optiek gebruikt verder gebruikt worden. Met andere woorden C++ staat dicht bij de machine. Van de andere kant is C++ bedoeld om de complexiteit van een probleem in kaart te brengen. Door een klassehiërarchie op te bouwen is het mogelijk om een klare kijk te behouden op de gegevens en bijbehorende acties in een probleem. Hierdoor is het mogelijk dat één persoon met C++ programma's van meer dan 25.000 regels kan ontwerpen en onderhouden. In C zou dit veel moeilijker zijn.

Waarom objectgeoriënteerd programmeren?

Veel C programmeurs blijven liever bij C en zijn niet geneigd om C++ te leren. Hun argumenten zijn dikwijls als volgt:

- C++ is een moeilijke taal. De concepten zijn veel abstracter en de mechanismen in de taal zijn op het eerste zicht niet erg duidelijk. Waarom een moeilijke taal zoals C++ gebruiken als het met eens simpele taal zoals C ook gaat.
- Bij C++ is het veel moeilijker om de programmauitvoering te volgen. Om de haverklap worden constructors uitgevoerd. Door het mechanisme van de late binding weet je niet direct waar je terecht komt bij de stap voor stap uitvoering.
- Sommigen willen toch de principes van het objectgeoriënteerd programmeren volgen, maar doen dit liever in C. Deze werkwijze is nogal omslachtig.
- Bij C++ is de programmeur verplicht om tijdens de analyse zijn denkproces aan te passen aan het object georiënteerd ontwerpen. Dit onvermijdelijke denkwerk gebeurt reeds vóór het programmeren en kan niet omzeild worden. In deze analysefase ben je verplicht om goed na te denken over gegevens en acties en hun onderling relaties. Het resultaat van dit denkwerk is het objectenmodel van het probleem. Dit model dient als leidraad tijdens het programmeerwerk. Tijdens de testfase zal dan blijken dat er minder denk- en programmeerfouten zijn en bijgevolg het programma sneller afgewerkt zal zijn.

Programmeerparadigma's

Tijdens de geschiedenis van de programmeertalen zijn er heel wat programmeerprincipes opgedoken. Eén van de eerste principes is het *programmeren met procedures*[27] Hierbij is het de bedoeling om alle acties die nodig zijn om een bepaalde taak uit te voeren worden in een routine, procedure of functie ondergebracht. De analyse die deze groepering vooraf gaat is vooral gericht naar uitvoerbare acties. Er ontstaat een hiërarchie van procedures.

[27] Procedure is synoniem van methode, functie of subroutine.

Voor het controleren van de volgorde waarin acties worden uitgevoerd werd eerst de goto gebruikt. Tegenwoordig wordt deze constructie niet meer gebruikt (tenzij in assembler). De if-then-else is er in de plaats gekomen. Dit principe heet *gestructureerd programmeren*. Deze stijl wordt alle moderne programmeertaal mogelijk gemaakt.

Na een aantal jaren is gebleken dat een goede data-analyse belangrijk is. Een aantal procedures die betrekking hebben op dezelfde gegevens worden in een module gegroepeerd. De data binnen de module mag niet rechtstreeks toegankelijk zijn, maar gebeurt door middel van procedures. Dit is het *modulair programmeren*. Hierbij wordt het principe van de *data-hiding* toegepast. De wijze waarop de data opgeslagen is, is niet gekend door de gebruiker van de module. Het modulair programmeren laat ook toe grotere programma's te ontwerpen. In C is een module een apart `.c` bestand waarin de gegevens in globale variabelen worden bijgehouden. Om de *data-hiding* mogelijk te maken moet het woord `static` aan elke declaratie voorafgaan. Deze methode heeft een nadeel: elke module stelt slechts één gegevensgroep voor, bijvoorbeeld een tabel met strings. Als we meerdere gegevensgroepen willen maken moeten we `struct` gebruiken. Hiermee verliezen we de *data-hiding*. De taal C is daarom niet geschikt om op een succesvolle manier gegevens voor de gebruiker te verbergen. Door dit nadeel is het beter om C te verlaten en C++ te gebruiken. C++ kent het principe van de klasse. Hiermee is een perfecte *data-hiding* mogelijk.

Bij het modulair programmeren ontstaan modules. Bij nieuwe projecten is het dikwijls lastig om bestaande modules opnieuw te gebruiken. Dikwijls zijn er kleine aanpassingen nodig of wordt toch maar de hele module herschreven. Het herbruiken van reeds bestaande programmatuur is niet gemakkelijk bij het *modulair programmeren*. Bij het *objectgeoriënteerd programmeren* is het mogelijk om afleidingen te maken van bestaande klassen. In de nieuwe klassen kunnen dan de kleine wijzigingen gebeuren. Een andere ontwerptechniek is het ontwerpen van een klasse waarin een aantal functies alleen maar als prototype voorkomen. In de afleiding van deze klassen wordt de implementatie van deze functies ingevuld. Door deze techniek is het mogelijk om een klasse te maken die nog algemeenheid als eigenschap hebben. Een algemene (of in OO termen: abstracte) klasse is niet bedoeld om hierin variabelen te declareren. Van een abstracte klasse worden afleidingen gemaakt en deze afleidingen worden gebruikt om variabelen te declareren. Door de techniek van de afleiding ontstaan er hiërarchieën van klassen. Die maken het mogelijk om de functionaliteit en gegevens in een groot programma goed in kaart te brengen. C programmeurs die klagen over de moeilijkheidsgraad van C++, klagen over het feit dat ze een OO analyse van gegevens/acties moeten uitvoeren; iets dat zij tot dan toe nooit deden. Daarom is C++ leren niet zomaar weer een nieuwe taal leren, het is een andere manier van denken.

Als besluit op deze inleiding geven we de tegenargumenten die vóór C++ pleiten:

- De concepten van een OO taal zijn abstracter omdat het denken tijdens een OO analyse abstracter is. Deze hogere graad van denken loont wel de moeite. Hiermee worden vroegtijdig de elementen (gegevens en bijbehorende acties) van het probleem gestructureerd en worden tegenstrijdigheden die later zouden kunnen opduiken bij onderhoud of wijziging vermeden.
- Het is inderdaad veel moeilijker om een C++ programma stap voor stap te volgen, want we worden geconfronteerd met alle details van alle klassen die doorlopen worden. Het is beter om niet stap voor stap te debuggen, maar wel op klasseniveau. Test elke klasse één voor één. Door de data-hiding is de relatie tussen de verschillende klassen minimaal. Elke klasse kan daarom als een op zich bestaand domein beschouwd worden.
- Het is beter om OO te programmeren met een taal die dit principe ondersteunt. Een goede programmeur kan wel in zekere mate OO principes in C toepassen, maar dit gaat gepaard met veel pointers naar functies, `void *` pointers en cast bewerkingen. Dit zijn allemaal constructies die gemakkelijk fouten introduceren. Bij een cast schakel je moedwillig de typecontrole door de compiler uit.
- Niet-objectgeoriënteerde verschillen tussen C en C++

Normaal gezien is het mogelijk om een C programma te compileren met behulp van een C++ compiler. Er zijn wel enkele kleine verschillen tussen beide compilers voor wat betreft de C syntax.

C constructies met een andere betekenis in C++

We geven hier een overzicht van de belangrijkste taal elementen die anders zijn in C++ dan in C.

- In C++ is er geen beperking op de lengte van namen.
- De C++ compiler moet eerst een prototype van een functie gezien hebben voor dat deze functie opgeroepen kan worden.
- Een karakterconstante is in C++ van het type `char`, in C is dit `int`.

Referentietype

Dit is een nieuw type in C++. Het referentietype[28] maakt het mogelijk om een variabele te declareren die als synomiem van een andere variabele dienst doet.

```
int a;
int &x = a;
int &y;            // fout: initialisatie ontbreekt
int &z = 17;       // fout: initalisatiewaarde moet een variabele zijn
                   //       waarvan het adres kan genomen worden
const int &q = 19; // ok: met const mag het wel

x = 5;             // a is nu 5
```

De variabele `a` is een gewoon geheel getal. De variabele `x` is een referentievariabele die verwijst naar de variabele `a`. Voor `x` is er geen opslagruimte voor een `int`. Als `x` gewijzigd wordt, wordt de inhoud van `a` gewijzigd. Een referentievariabele moet geïnitialiseerd worden bij de declaratie. Daarom is de declaratie van `y` fout. De initialisatie van `z` is fout; `17` is een constante waarvan geen adres kan genomen kan worden en dat mag niet. `17` is een *rvalue* en `a` is een `lvalue`; met de laatste mag je dus wel initialiseren.

Het is niet mogelijk om na de declaratie de referentievariabele naar een andere variabele te laten verwijzen. Als je dat echt wilt, moet je opnieuw pointers gebruiken.

De referentievariabele wordt dikwijls als formele parameter gebruikt. Hier is een klassiek voorbeeld:

```
void swap(int &x, int &y)
{
   int h;

   h = x;
   x = y;
   y = h;
}

int main()
{
   int a = 5;
   int b = 6;

   swap (a, b);   // verwissel a en b

   return 0;
}
```

Bij het gebruik als parameter gedraagt een referentietype zich als een `VAR` parameter in Pascal. Het voordeel is het feit dat in de functie `swap` de variabelen `x` en `y` er als een gewone variabele uitzien en niet als een pointervariabele. Dit is dus *call by reference*.

Het referentietype komt later opnieuw aan bod wanneer we de operatoren bespreken.

[28] Dit referentietype heet voluit eigenlijk het *lvalue referentietype*. In C++11 heb je nu ook het *rvalue referentietype*.

De klasse in C++

Een klasse definiëren

C++ kent niet alleen het type `struct` maar ook het type `class`. Beiden kunnen gebruikt worden om gegevens in te kapselen. Dit doen we om gegevens, die sterk verbonden zijn, samen te brengen onder een noemer. Het sleutelwoord `class` is nieuw in C++ en laat de bescherming van de gegevens binnen een klasse toe.

Het eerste voorbeeld heeft wat te maken met grafische weergave van gegevens. In een grafische omgeving moeten we de coördinaten van een punt op het scherm bijhouden. Dit doen we door de coördinaten van een plaats op te slaan in een klasse:

```
class Punt
{
   int x;
   int y;
};
```

De klasse `Punt` bevat dus de velden `x` en `y`. Deze vorm van groeperen kennen we al van C. Met de klasse `Punt` kan een variabele gedeclareerd worden en we kunnen trachten om de velden `x` en `y` te bereiken.

```
Punt p1;
```

De variabele `p1` is van het type `Punt` en is in staat om 2 coördinaten te onthouden. Dit is niet meteen zichtbaar aan de variabele. Dit is het principe van de *inkapseling*. We gebruiken de term *object* om een variabele van een zekere klasse aan te duiden. Als we het in de toekomst over objecten hebben, dan bedoelen we hiermee variabelen of stukken dynamisch geheugen waarin zich informatie van een zekere klasse bevindt.

Toegang tot leden

Als we de leden van het object `p1` willen bereiken, dan zouden we het volgende kunnen uitproberen:

```
int main()
{
   Punt p1;

   p1.x = 50;
   p1.y = 70;

   return 0;
}
```

Helaas geeft dit programma twee compilatiefouten. De twee velden `x` en `y` zijn niet toegankelijk van buiten het object. Als we binnen de definitie van de klassen niet één van de woorden `private`, `protected` of `public` zijn alle leden privaat, `private` dus. We hadden evengoed dit kunnen schrijven:

```
class Punt
{
private:
   int x;
   int y;
};
```

Op deze wijze wordt duidelijk weergegeven dat de leden `x` en `y` niet publiek toegankelijk zijn.

Een constructor bijvoegen

Omdat de leden van de klasse `Punt` niet publiek toegankelijk zijn is er een probleem om bijvoorbeeld de variabele `p1` te initialiseren of om de waarden `x` en `y` van een `Punt` object te weten te komen. Daarom zijn we verplicht tot de leden van een klasse via *lidfuncties* te organiseren. Een *lidfunctie* is een functie die deel uitmaakt van een klasse. Dikwijls wordt ook de term *methode* voor dit soort functies gebruikt. Er is

een speciale vorm van een lidfunctie die enkel voor de initialisatie van een object wordt gebruikt. Deze vorm wordt *constructor* genoemd. Een constructor krijgt als naam de naam van de klasse.

```
class Punt
{
private:
   int x;
   int y;

public:
   Punt(int ix, int iy);
};
```

Binnen de klassebeschrijving noteren we een prototype van een functie. Omdat we voor de functienaam de naam van de klasse kiezen, is deze functie een constructor. We voorzien de constructor van twee formele parameters `ix` en `iy`. Dit betekent dat we aan de constructor twee getallen kunnen meegeven die dienen voor de initialisatie van het object. Bij een constructor mogen we nooit een return type noteren; parameters mogen wel. In dit voorbeeld zijn er twee parameters.

Wat deze constructor moet doen, is nog niet vastgelegd. Binnen de klasse staat alleen maar een prototype. Nu zijn er twee manieren om de implementatie vast te leggen.

Een constructor implementatie buiten de klasse

We noteren de implementatie van de constructor buiten de accolades van de klassebeschrijving. Om aan te geven dat het hier gaat over een lidfunctie van de klasse `Punt` moeten we nog eens de klassenaam aan de functienaam laten voorafgaan. Tussen de klassenaam en de functienaam schrijven we twee dubbele punten.

```
Punt::Punt(int ix, int iy)
{
   x = ix;
   y = iy;
}
```

Omdat het hier om een constructor gaat, schrijven we tweemaal `Punt` . Eenmaal als klassenaam en als functienaam. Binnen de acties van een lidfunctie zijn de dataleden van een klasse vrij toegankelijk. De namen `x` en `y` binnen de constructor zijn de twee dataleden van een `Punt` object. Aan elk van de dataleden wordt een startwaarde toegekend.

Voor het toekennen van een startwaarde is er bij constructors aan andere schrijfwijze mogelijk. Hierbij wordt na een dubbele punt de lijst van te initialiseren datavelden geschreven met telkens de startwaarde erbij:

```
Punt::Punt(int ix, int iy) : x(ix), y(iy)
{
}
```

Deze schrijfwijze mag alleen maar bij constructors toegepast worden.

Een constructor implementatie binnen de klasse

We kunnen de implementatie van de constructor ook binnen de klasse noteren. Het voordeel is dat we minder schrijfwerk hebben. Het nadeel is dat als we de implementatie van de constructor wijzigen, dan moet er binnen de klassedefinitie gewijzigd worden. Vermits we alle klassedefinities in headerbestanden onderbrengen, veroorzaakt dit de hercompilatie van alle `.cpp` bestanden die van deze klasse gebruik maken.

```
class Punt
{
private:
   int x;
   int y;
```

```
public:
   Punt(int ix, int iy) : x(ix), y(iy)
   {
   }
};
```

Voor een constructor maakt het geen verschil uit welke twee opties gekozen wordt. Voor lidfuncties, die dus geen constructor zijn, is er wel een verschil tussen implementatie binnen of buiten de klasse. Dit onderscheid bespreken we later.

Objecten declareren

Zoals we met een eenvoudig type een variabele kunnen declareren, kunnen we met een klasse een object declareren.

```
#include <iostream>

Punt pg(23, 34);

int main()
{
   Punt p1(30, 40);
   Punt p2 = Punt(44, 55);

   std::cout << "main\n";

   return 0;
}
```

Omdat er een constructor is die twee gehele getallen verwacht als parameter, moeten we bij de initialisatie tussen ronde haken twee getallen voorzien. Het is hier dat de constructor in actie treedt. Een constructor kunnen we nooit zelf starten. Een constructor wordt uitgevoerd voor een object, zodra dit object tot leven komt. Voor `pg` is dit zelfs voor de start van `main()`. Dit betekent dat voor het starten van `main()` de dataleden `x` en `y` van het object `pg` met `23` en `34` worden gevuld. Daarna start`main()` en daarna wordt achtereenvolgens de constructor voor `p1` en `p2` opgeroepen. Dan pas start de tekstuitvoer met `std::cout`. Voor `p1` en `p2` is telkens een andere schrijfwijze van de initialisering toegepast.

Tekstuitvoer in C++ doen we zo:

```
#include <iostream>

std::cout << "main\n";
```

`std::cout` is het standaard uitvoerkanaal. `std::cout` is een voorgedefiniëerde variabele en bevindt zich binnen de `std` naamruimte. In de recente C++ schrijfstijl wordt de `std::` naamruimte geschreven waar die nodig is.

Meerdere constructors

Het is mogelijk om meerdere constructors te voorzien binnen een klasse. Het aantal en het type parameters van alle constructors moeten verschillend zijn. Met andere woorden: elke constructor heeft een ander prototype. In de klasse `Punt` hadden we al een constructor met als parameters de coördinaten van een punt. We voegen nu een constructor bij die geen parameters heeft.

```
class Punt
{
private:
   int x;
   int y;

public:
```

```
   Punt(int ix, int iy) : x(ix), y(iy)
   {
   }
   Punt() : x(0), y(0)
   {
   }
};
```

De implementatie van deze nieuwe constructor plaatsen we voor het gemak in de klasse zelf. Deze constructor zorgt ervoor de dataleden `x` en `y` automatisch 0 worden als we de expliciete initialisatie weglaten bij de declaratie van een object. In de volgende versie van `main` zien we twee objecten van de klasse `Punt`; één met en één zonder initialisatie.

```
int main()
{
   Punt p1(30, 40);
   Punt p2;

   return 0;
}
```

Het object `p1` wordt met de eerste constructor geïnitialiseerd en `p2` wordt geïnitialiseerd met de tweede constructor. Dit betekent dat de `p2.x` en `p2.y` allebei nul worden. De compiler beslist aan de hand van het aantal en het type actuele parameters welke constructor bij de declaratie opgeroepen wordt. Dit is meteen ook de reden waarom er geen twee constructors met hetzelfde prototype mogen zijn.

Als algemene regel geldt dat een methode- of functienaam meerdere malen mag gebruikt worden op voorwaarde dat het type en/of het aantal parameters verschilt. Dit verschil moet er zijn opdat de compiler de oproep van de juiste methode kan bepalen.

Een object initialiseren door een ander object

Het is mogelijk om bij de declaratie een object te initialiseren met een ander object.

```
int main()
{
   Punt p1(30, 40);
   Punt p2 = p1;

   return 0;
}
```

We zouden kunnen verwachten dat de compiler zou eisen dat er een constructor met prototype

```
Punt(const Punt &x);
```

voorkomt in de klasse. We hoeven deze constructor echter niet te definiëren omdat dit automatisch gebeurt. Elke klasse krijgt automatisch een constructor die dient voor de initialisatie met een object van dezelfde klasse. Deze constructor wordt *copy constructor* genoemd; hij kopieert één voor één alle dataleden van het ene naar het andere object. Dit gedrag is hetzelfde als bij het kopiëren van structuren in C. Bij de klassen `Punt` is het lidsgewijs kopiëren het juiste gedrag. In het bovenstaande voorbeeld worden de datavelden `x` en `y` van `p1` naar `p2` gekopieerd.

Objecten kopiëren

Zo kunnen we ook objecten kopiëren met een toekenning. Deze operator kopieert zoals bij de *copy constructor* alle dataleden.

```
int main()
{
   Punt p1(30, 40);
   Punt p2;
```

```
   p2 = p1;

   return 0;
}
```

Voor deze bewerking heeft de compiler automatisch een operator voor de `=` bewerking binnen de klasse bijgevoegd. Deze copy constructor heeft `Punt &operator=(const Punt &pt)`. We zien in deze notatie het referentietype. Het bijvoegen van bewerkingen zoals `operator=()` bekijken we later nog wel en dan volgt de uitleg waarvoor in deze notatie het referentietype nodig is.

Lidfuncties in een klasse bijvoegen

De twee dataleden van de klasse `Punt` zijn privaat. Dit betekent dat we niet rechtstreeks toegang krijgen tot de dataleden. Daarom voegen we een klassefunctie bij die de coördinaten van een `Punt` op het scherm drukt.

```
class Punt
{
private:
   int x;
   int y;

public:
   Punt(int ix, int iy) : x(ix), y(iy)
   {
   }
   Punt() x(0), y(0)
   {
   }
   void druk();
};

void Punt::druk()
{
   std::cout << "<" << x << "," << y << ">" << std::endl;
}
```

De klassefunctie druk kan zonder meer toegang krijgen tot de dataleden van het object in kwestie. De functie `druk()` kan alleen maar gestart worden met een concreet object:

```
void main()
{
   Punt p1(67,78);
   Punt p2(34,98);

   p1.druk();
   p2.druk();
}
```

De notatie van de oproep van een klassefunctie is dezelfde als in C voor de toegang tot een veld van een structuur. De naam van het object wordt gevolgd door een punt en de naam van de klassefunctie.

Inline uitvoering van een lidfunctie

Het is mogelijk om de tijd die nodig is voor de oproep en de terugkeer van een functie te elimineren. Dit is nodig als een klassefunctie zeer kort is.

We gaan twee lidfuncties aan de klasse `Punt` bijvoegen om de waarden van dataleden `x` en `y` terug te geven. We tonen twee versies van de implementatie van de lidfuncties.

Een lidfunctie implementatie buiten de klasse

Binnen de klasse `Punt` worden twee prototypes voor `haalx` en `haaly` bijgevoegd.

```
class Punt
{
private:
   int x;
   int y;

public:
   Punt(int ix, int iy) : x(ix), y(iy)
   {
   }
   Punt() : x(0), y(0)
   {
   }
   void druk();
   int haalx();
   int haaly();
};
```

De implementaties van de twee functies ziet er zo uit:

```
int Punt::haalx()
{
   return y;
}

int Punt::haaly()
{
   return y;
}
```

We kunnen deze twee functies expliciet gebruiken om de waarden van de coördinaten op te halen zonder grens van de inkapseling te overtreden. Dit wordt als volgt gedaan:

```
int main()
{
   Punt pp(67, 89);

   std::cout << "<" << pp.haalx()
        << "," << pp.haaly() << ">" << std::endl;

   return 0;
}
```

Bij deze implementatie is er sprake van een echte subroutine. Er is bijgevolg een oproep en een terugkeer. Het nadeel van deze twee functies is dat ze zeer kort zijn; dit betekent dat er meer tijd besteed wordt aan de oproep en de terugkeer (`jsr` en `ret` instructies) dan aan de uitvoering van de acties van de functies. Daarom kan gekozen worden voor de inline uitvoering van dit soort van korte functies.

Een lidfunctie implementatie binnen de klasse

We plaatsen de opdrachtregel van de klassefuncties binnen de definitie van de klasse.

```
class Punt
{
private:
   int x;
   int y;

public:
```

```
   Punt(int ix, int iy)
   {
      x = ix;
      y = iy;
   }
   Punt()
   {
      x = 0; y = 0;
   }
   void druk();
   int haalx()
   {
      return x;
   }
   int haaly()
   {
      return y;
   }
};
```

Door deze notatievorm wordt de functie *inline* uitgevoerd. Elke oproep in C++ notatie wordt vervangen door de instructies van de opdrachtregel. Het is evident dat deze oplossing alleen efficiënt is bij zeer korte klassefuncties. Het gevolg is wel dat het programma in zijn totale lengte (in machineinstructies uitgedrukt) langer wordt omdat het principe van de subroutine niet wordt toegepast.

Tot slot maken we nog de opmerking dat in alle voorbeelden de variabelen meteen hun geheugenruimte toegewezen kregen door de declaratie. Als een `Punt` bestaat uit tweemaal `int` en een `int` bestaat elk uit 4 bytes, dan zal de variabele `Punt p` 8 bytes in beslag nemen. Je ziet dus dat vastleggen van de geheugenruimte op dezelfde wijze gebeurt voor objecten van een klasse en enkelvoudige types zoals `int`. Dit is anders dan in Java. In Java worden objecten altijd dynamisch in de heap gereserveerd. In C++ is er de keuze:

```
Punt  p1;
Punt *p2 = new Punt(5, 6);
```

Hier krijgt `p1` geheugenruimte toegewezen door de compiler. Bij `p2` gebeurt dat pas bij de uitvoering van het programma (zoals in Java). C++ zou C++ niet zijn als we voor het maken van objecten geen meerdere mogelijkheden zouden hebben.

Bewerkingen in een klasse

C++ kent de mogelijkheid om een nieuwe betekenis te geven aan een bewerkingsteken afhankelijk van de klasse waarop de bewerking betrekking heeft. Zo kan men een andere betekenis geven aan de optelling bij breuken en bij complexe getallen. We geven een voorbeeld dat handelt over breuken.

Bewerkingen als functies in een klasse

We ontwerpen de klasse om een breuk op te slaan. De klasse krijgt twee dataleden: één voor de teller en één voor de noemer. Beide worden opgeslagen als een geheel getal. De constructor is voorzien van verstekwaarden; zo krijgt een breuk de waarde 0/1 wanneer de initialisatiewaarden ontbreken.

```
// breuk.h
class Breuk
{
public:
   Breuk(int t=0, int n=1) : teller(t), noemer(n)
   {
   }
   Breuk &operator++();
   Breuk operator+(Breuk b);
```

```
private:
   int teller;
   int noemer;
};
```

Wanneer we een bewerkingsteken willen definiëren dan geven we de bijbehorende functie een speciale naam. We combineren het sleutelwoord `operator` met het bewerkingsteken in kwestie. Voor de `++` bewerking wordt dit:

```
Breuk &operator++();
```

Deze functie geeft een `Breuk` als referentietype terug. Dit is nodig omdat de `++` bewerking in uitdrukkingen kan voorkomen. Het terugkeertype is het referentietype omdat de `++` bewerking zowel links of rechts van een toekenning kan voorkomen.

Dit betekent dat de de volgende toekenning mogen schrijven. Dit voorbeeld zullen we zelf niet zo gauw schrijven omdat ze slecht leesbaar is. Vooral de `++` links van het `=`-teken schept onduidelijkheid. Hier gaat het om de betekenis van het referentietype: dit type stelt een waarde/notatie voor die wijzigbaar is.

```
Breuk b1;
Breuk b2;

b1++ = b2++;
```

Op dezelfde wijze wordt de functienaam voor de optelling samengesteld:

```
Breuk operator+(Breuk b);
```

In dit geval is er een parameter: dit is de breuk die bij andere breuk wordt opgeteld. Het resultaat van de bewerking is het `Breuk` type. De bewerking kan zo gebruikt worden:

```
b3 = b1 + b2;
```

We zouden de bewerking ook als een functie kunnen starten:

```
b3 = b1.operator+(b2);
```

Deze schrijfwijze is alleen maar nuttig om te zien hoe de bewerking gestart wordt.

De implementatie van beide bewerkingen worden als gewone klassefuncties geschreven. Voor de `++` bewerking wordt de noemer éénmaal bij de teller opgeteld. Met

```
return *this;
```

wordt een referentie naar het huidige object als resultaat teruggegeven.

```
// breuk.cpp
#include <iostream>
#include "breuk.h"

Breuk &Breuk::operator++()
{
   teller += noemer;
   return *this;
}

Breuk Breuk::operator+(Breuk b)
{
   Breuk nb;

   nb.teller = teller * b.noemer + noemer * b.teller;
   nb.noemer = noemer * b.noemer;

   return nb;
}
```

In de + bewerking wordt een nieuwe `Breuk` gemaakt. De som van de twee op te tellen breuken wordt in deze nieuwe variabele geplaatst. Met `return`, in dit geval de som, gaat het resultaat terug naar de oproeper. Hier wordt geen referentietype teruggegeven omdat we de waarde van een lokale variabele teruggeven. Als we hier het referentietype zouden gebruiken, geven we een wijzigbare lokale variabele terug. Dit mag niet omdat deze variabele niet meer bestaat als de functie stopt.

Vriendfuncties van een klasse

In sommige gevallen is het nodig om een bewerking als een functie buiten de klasse te definiëren. Dit is het geval bij de functie die de uitvoer van een `Breuk` verzorgt. Omdat deze functie toegang moet krijgen tot de private leden van de klasse `Breuk` maken we de functie een vriend van de klasse `Breuk`.

```
friend ostream &operator<<(ostream &os, Breuk b);
```

Het bovenstaande prototype wordt in de klasse bijgevoegd. Met het woord `friend` gevolgd door een prototype wordt aangegeven dat een niet-klasse functie toegang krijgt tot alle private leden.

Voor de uitvoer wordt het naar links schuif teken « gebruikt. We schrijven dit teken na het woord `operator`. De parameters van de uitvoerbewerking zijn het uitvoerkanaal en de `Breuk` die getoond moet worden. `ostream` is het type van het uitvoerkanaal. `std::cout` behoort tot dit type. Als terugkeertype zien we een referentie naar `ostream`. We geven het uitvoerkanaal terug als referentie. Dit is nodig omdat de uitvoerbewerking samen met het uitvoerkanaal en de breuk opnieuw als een `ostream` aanzien wordt. Hierdoor kan men meerdere uitvoerbewerkingen na elkaar schrijven.

```
(std::cout << b1) << b2;
```

In de bovenstaande uitvoer wordt `std::cout << b1` opnieuw als een uitvoerkanaal aanzien. Naar dit kanaal wordt de uitvoer van `b2` gestuurd.

De implementatie van de uitvoerbewerking gaat na of de noemer 1 is. Indien ja, wordt alleen de teller getoond. Anders worden teller en noemer gescheiden met een deelstreep getoond.

```
ostream & operator<<(ostream &os, Breuk b)
{
   if (b.noemer == 1)
   {
      std::cout << b.teller;
   }
   else
   {
      std::cout << b.teller << "/" << b.noemer;
   }
   return os;
}
```

De uitvoerbewerking geeft als resultaat het doorgegeven uitvoerkanaal terug.

We tonen nog een voorbeeld van een hoofdprogramma waarin de klasse `Breuk` gebruikt wordt.

```
#include <iostream>
#include "breuk.h"

void main()
{
   Breuk b;

   std::cout << b << std::endl;
   b++;
   std::cout << b << std::endl;

   Breuk c(1,4);
   Breuk d(1,2);
   Breuk e;
```

```
    e = c + d;
    std::cout << e << std::endl;
}
```

Lvalue en rvalue referenties

Tot nu toe hebben we enkel de notatie `Breuk&` gebruikt als referentietype. Sinds de komst van C++11 is er nog een tweede referentietype bijgekomen dat als `Breuk&&` geschreven wordt. Dit nieuwe type wordt het *rvalue referentietype* genoemd. Het reeds bestaande type wordt dan *lvalue referentietype* genoemd. Het nieuwe referentietype is ingevoerd om nog een betere optimalisaties te verkrijgen die vermijden dat we geheugen moeten kopiëren wanneer we in een methode een waarde teruggeven. In het C++11 deel van deze cursustekst zal dit nieuwe referentietype gedetaileerd uitgelegd worden.

Dynamische objecten

Zoals C kent C++ ook het principe van het dynamisch reserveren van geheugen voor gegevensopslag. In C++ zijn voor dit doel de operatoren `new` en `delete` ingevoerd.

Het voordeel van het dynamisch reserveren van geheugen (in C++ bijna altijd geheugen voor objecten) is de grotere flexibiliteit. Je bepaalt zelf wanneer je objecten moet bijmaken en hoelang deze objecten mogen blijven bestaan. Als je objecten niet dynamisch reserveert, blijven ze ofwel altijd bestaan als het globale variabelen zijn of bestaan ze maar even als het lokale variabelen binnen een functie of methode zijn.

De `new` bewerking

Met de `new` bewerking kan geheugen op dynamische wijze gereserveerd worden. In tegenstelling tot C waar `malloc()` een ingebouwde functie is, is in C++ `new` een ingebouwde bewerking. Deze bewerking wordt toegepast op de typeinformatie.

`new` bij een niet-klasse

De eerste vorm waarin `new` gebruikt kan worden is de toepassing op een enkelvoudig type. Als we bijvoorbeeld geheugen voor één `int` willen reserveren dan kan dit zo:

```
int *pi = new int;

*pi = 5;
```

De toepassing van de bewerking `new` op het type levert het adres op van een blokje geheugen. In dit geheugen is plaats voor één getal van het type `int`. In tegenstelling tot C is er geen cast-bewerking nodig.

`new` bij een klasse

Dikwijls wordt de `new` bewerking gebruikt om geheugen te reserveren voor objecten. We gebruiken dan als type-informatie de klassenaam. In het volgende voorbeeld wordt een object van de klasse `Punt` gereserveerd.

```
Punt *pu = new Punt;
```

ofwel, in een andere schrijfwijze:

```
Punt *pu;

pu = new Punt;
```

Bij het uitvoeren van de `new` bewerking gebeuren er eigenlijk twee stappen:

- `new` reserveert zoveel geheugen als nodig is voor de klasse.
- Indien de reservatie gelukt is, wordt nog de constructor uitgevoerd.

In het voorgaande voorbeeld wordt de constructor zonder parameter uitgevoerd. Hierdoor worden de dataleden allebei nul.

Bij de klasse `Punt` is het mogelijk om bij het dynamisch reserveren van een object meteen ook gegevens voor de initialisatie mee te geven. We maken dan gebruik van de constructor met twee parameters.

```
void fu()
{
   Punt *pa;
   Punt *pb;

   pa = new Punt(23,34);
   pb = new Punt(45,56);

   pa->druk();
   pb->druk();
}
```

Als in het bovenstaande voorbeeld de functie `fu()` gestart wordt, worden er twee objecten in dynamisch geheugen gereserveerd. Omdat bij `new` na de klassenaam twee getallen voorkomen, wordt de constructor met twee parameters gestart. Na het reserveren van twee objecten wordt met de methode `druk()` de coördinaten in `pa` en `pb` op het scherm geschreven. Omdat `pa` en `pb` pointers zijn, moet een pijl gebruikt worden om methoden te bereiken.

De `delete` bewerking

Als in het voorgaande voorbeeld het einde van de functie bereikt wordt, houden de pointers `pa` en `pb` op te bestaan. Vermits ze allebei wijzen naar dynamisch gereserveerd geheugen, zou hierdoor een geheugenlek ontstaan. Daarom moet vóór het einde van de functie het geheugen vrijgegeven worden. Dit doen we met de `delete` bewerking.

```
void fu()
{
   Punt *pa;
   Punt *pb;

   pa = new Punt(23,34);
   pb = new Punt(45,56);

   pa->druk();
   pb->druk();

   delete pa;
   delete pb;
}
```

Na het woord `delete` schrijven we de naam van de pointervariabele die wijst naar het dynamisch geheugen. Voor elke `new` bewerking die in een programma voorkomt, moet er een overeenkomstige `delete` bewerking zijn.

`new` en `delete` bij arrays

Bij het gebruik van de bewerking `new` bestaat de mogelijkheid om geheugen voor arrays te reserveren. We schrijven dan na `new` een arraytype. De waarde tussen de rechte haken mag wel een variabele zijn. Hierdoor kan de lengte van de array dynamisch bepaald zijn. In het volgende voorbeeld krijgt `p` het adres van een blok van 100 char's.

```
char *p = new char[100]
delete [] p;
```

Bij het vrijgeven van het geheugen met `delete` moet aangegeven worden dat het gaat over een array. Daarom moeten voor de variabelenaam rechte haken geschreven worden. De vrijgave van het geheugen

moet expliciet geschreven worden voordat de pointervariabele ophoudt te bestaan. Als je dat niet doet, krijg je geheugenlekken[29].

Toch moeten we even opmerken dat de rechte haken `[]` nodig zijn om ervoor te zorgen dat voor elk element van de array de destructor correct gestart wordt. In het bovenstaande voorbeeld is `char` het inhoudstype en speelt het geen rol of we de rechte haken al dan niet schrijven. Wanneer we het geheugen zo zouden reserveren:

```
Punt *pu = new Punt[100]
delete [] pu;
```

moeten we de rechte haken zeker schrijven zodat de destructor van `Punt` zeker loopt voor elk van de 100 punten.

Het gebruik van `new` binnen een klasse

Het is mogelijk om de hoeveelheid geheugen die nodig is binnen de klasse ook dynamisch te reserveren. Op deze manier zijn er geen beperkingen op de lengte van de binnen een klasse opgeslagen gegevens.

In het volgende voorbeeld wordt een klasse `Tekst` gedemonstreerd. Deze klassen kan gebruikt worden om tekstobjecten te creëren. Om te vermijden dat er conflicten ontstaan als er lange teksten opgeslagen moeten worden, is de opslag van de string binnen een object dynamisch[30]. De klasse `Tekst` ziet er als volgt uit:

```
class Tekst
{
private:

   char *ptekst;

public:
   Tekst(const char *pv = "");
   ~Tekst();
   char *str() const
   {
      return( ptekst );
   }
   Tekst &operator=(const Tekst &t);
   Tekst &operator+=(const Tekst &t);
   Tekst  operator+(const Tekst &t);
};
```

Voor de opslag van de string binnen het object is er het private datalid `ptekst`. De constructor

```
Tekst(const char *pv = "");
```

wordt opgeroepen als een `Tekst` object met een `char` string initialiseren. Indien de parameter bij de oproep van de constructor ontbreekt, dan wordt een lege string als verstekwaarde gebruikt.

De implementatie van de constructor is als volgt:

```
Tekst::Tekst(const char *pv)
{
   ptekst = new char [strlen(pv) + 1];
   strcpy(ptekst, pv);
}
```

Met `new` worden zoveel bytes gereserveerd als de string lang is. Er is ook een extra byte voor de nulwaarde op het einde. Het resultaat van `strlen()` is immers de lengte van de string zonder de eindnul meegerekend. De originele string wordt in het gereserveerde geheugen gekopieerd. De kopieerbewerking is nodig om

[29] Met de tool `valgrind` kan je mogelijke geheugenlekken opsporen.

[30] In dit voorbeeld wordt de opslag met een `char *` string georganiseerd. Er worden ook C stringfuncties zoals `strlen()` gebruikt. Als je C en C++ vakkundig combineert, is er geen probleem. Uiteraard dient `Tekst` enkel als voorbeeld, in praktijk gebruik je beter `std::string`.

ervoor te zorgen dat object een eigen string in *eigendom* heeft. Indien we alleen het adres van de string zouden kopiëren, dan ontstaat er een situatie waarin een object verwijst naar geheugen die niet door het object wordt beheerd. Dit zou gevaarlijke situatie zijn.

Een destructor bijvoegen

Omdat er in de constructor dynamisch geheugen wordt gereserveerd, is het nodig dat in de klasse ook een destructor bestaat. Het prototype wordt met een tilde geschreven:

```
~Tekst();
```

Een destructor heeft geen parameters en geen terugkeertype. Een destructor kan wel virtueel zijn (een constructor daarentegen niet). Ook in de implementatie komt de tilde voor.

```
Tekst::~Tekst()
{
   std::cout << "delete " << ptekst << std::endl; // alleen voor test
   delete [] ptekst;
}
```

In deze destructor wordt met `delete` het geheugen van de string vrijgegeven. De uitvoerbewerking staat er alleen maar om te kunnen zien wanneer de destructor uitgevoerd wordt en is daarom niet noodzakelijk.

De klasse `Tekst` gebruiken

Het gebruik van de klasse `Tekst` is als volgt:

```
#include <iostream>
#include "tekst.h"

int main()
{
   Tekst t("hallo");

   std::cout << t.str() << std::endl;

   return 0;
}
```

De definitie van de klasse plaatsen we best in een headerbestand. Zo bevindt de definitie van `Tekst` zich in het bestand `tekst.h`. De tekst `t` wordt geïnitialiseerd met de string `"hallo"`. Vlak voor het einde van `main`() wordt de destructor opgeroepen voor het object `t`. Met de methode `str()` verkrijgen we het adres van de eerste `char` van de opgeslagen tekst. De klasse `Tekst` kunnen we gebruiken zonder dat we iets zien van de wijze waarop de implementatie binnen de klasse gemaakt is. Deze inkapseling is één van de principes van het objectgeoriënteerd programmeren.

Voor de klasse `Tekst` betekent dit dat alle problemen met te kleine buffers vermeden worden. In andere woorden, het gehele geheugenbeheer voor de opslag van de tekst is in de klasse ingekapseld.

Bewerkingen in een klasse bijvoegen

In het voorgaande voorbeeld is de klasse `Tekst` eerder beperkt. Daarom voegen we een tweetal bewerkingen bij in de klasse. We zouden deze bewerkingen kunnen bijvoegen in de vorm van klassefuncties zoals bijvoorbeeld `druk()` in de klasse `Punt`. Een ander alternatief is het veranderen van de betekenis van bewerkingstekens binnen een klasse. Dit betekent dat een bewerkingsteken een nieuwe betekenis krijgt. Volgens dit principe gaan we het `=` teken, `+=` teken en het `+` teken koppelen aan een klassefunctie binnen de klasse `Tekst`. Binnen de definitie van de klasse `Tekst` worden de prototypes voor deze twee bewerkingen bijgevoegd.

```
Tekst &operator=(const Tekst &t);
Tekst &operator+=(const Tekst &t);
Tekst operator+(const Tekst &t);
```

Het zijn twee klassefuncties met een speciale naam. We laten het woord `operator` volgen door het bewerkingsteken dat we een nieuwe betekenis willen geven. De klassefuncties `operator=`, `operator+=` en `operator+` hebben één formele parameter. Via deze parameter wordt de rechteroperand van de bewerking doorgegeven. De linkeroperand wordt doorgegeven via de impliciete pointer naar het object. Met de plusoperator kunnen we dan schrijven:

```
Tekst t1("dag ");
Tekst t2("wereld");
Tekst t3("");

t3 = t1 + t2;
```

Deze 'optelling' zou ook als volgt geschreven kunnen worden:

```
t3 = t1.operator+(t2);
```

Deze schrijfwijze is niet zo goed leesbaar, maar geeft wel duidelijk weer hoe de twee operands aan de optelling worden doorgegeven. De implementaties van de `=` en `+=` bewerkingen zien er zo uit:

```
Tekst &Tekst::operator=(const Tekst &t)
{
   delete [] ptekst;                          // verwijder de oude tekst
   ptekst = new char [strlen(t.str() ) + 1]; // ruimte voor nieuwe tekst
   strcpy(ptekst, t.str() );                  // kopieer tekst
   return( *this );
}

Tekst &Tekst::operator+=(const Tekst &t)
{
   char *poud;

   poud = ptekst;// hou oude tekst opzij

// reserveer ruimte voor nieuwe tekst
   ptekst = new char [strlen(poud) + strlen(t.str() ) + 1];

   strcpy(ptekst, poud);// kopieer eerste tekst
   strcat(ptekst, t.str() ); // voeg tweede tekst erbij
   delete poud;// verwijder oude tekst
   return *this;
}
```

Bij elk van de twee bewerkingen wordt opnieuw dynamisch geheugen gereserveerd omdat de lengte van de nieuwe tekst, die in een object opgeslagen wordt, groter kan zijn dan de oude tekst. Telkens wordt het betrokken object via `return` teruggegeven. Dit is nodig omdat het resultaat van de bewerking ook van het type `Tekst` is.

De `+` bewerking kan kort geschreven worden:

```
Tekst Tekst::operator+(const Tekst &t)
{
   Tekst nt;   // nieuwe tekst

   nt = *this; // kopieer eerste tekst
   nt += t;    // voeg tweede tekst bij
   return nt;  // geef nieuwe tekst terug als resultaat
}
```

We maken gebruik van een lokaal `Tekst` object `nt`. Met een toekenning en daarna `+=` bewerking worden de twee bronteksten samengevoegd in een nieuwe tekst. Deze nieuwe tekst wordt als resultaat teruggegeven. In de `+` bewerking wordt gebruik gemaakt van de eerder ontworpen `=` en `+=` bewerkingen. Het terugkeer type is in dit geval géén referentietype.

Bewerkingen in een klasse gebruiken

Het gebruik van de klasse `Tekst` is als volgt:

```
#include <iostream>
#include "tekst.h"

int main()
{
   Tekst t("hallo");
   Tekst t2;
   Tekst t3("o");

   std::cout << t.str() << std::endl;
   t2 = t;
   t2 = t + t3;
   std::cout << t2.str() << std::endl;

   return 0;
}
```

Merk op dat er een verschil is tussen de twee bewerkingen met het `=` teken in het volgende fragment:

```
{
   Tekst ta("1234");
   Tekst tb = ta;
   Tekst tc("");

   tc = ta;
}
```

Bij het eerste `=` teken wordt de copyconstructor gestart om `tb` te initialiseren; bij het tweede `=` teken wordt de klassefunctie `operator=()` gestart. Let op: de copyconstructor hebben we niet zelf bijgevoegd in de klasse `Tekst`. Daarom wordt de default copyconstructor uitgevoerd. Deze is evenwel niet geschikt voor gebruik van zodra binnen een klasse zelf dynamisch geheugen wordt bijgehouden. Dit is de reden waarom het bovenstaand fragment problemen kan geven zolang geen eigen versie van de copyconstructor binnen de klasse `Tekst` wordt bijgevoegd.

Objecten binnen objecten

In vele gevallen is het nuttig om een klasse te beschouwen als een enkelvoudig type. We gaan dan gemakkelijker klassen gebruiken om daarmee nieuwe klassen samen te stellen. De techniek die we nu voorstellen is het gebruik van een bestaande klasse als type voor dataleden van een nieuwe klasse. Het voorbeeld, dat we geven, heeft te maken met lijnen. Als we de gegevens van een lijn willen bijhouden, dan moeten we het begin- en eindpunt van de lijn opslaan. Een lijn bestaat uit twee punten of anders gezegd: de klasse `Lijn` bevat twee dataleden van de klasse `Punt`. Dit principe wordt aggregatie genoemd. We demonstreren dit met een voorbeeld.

```
#include <iostream>
#include <math.h>
#include "punt.h"


class Lijn
{
private:
   Punt p1;
   Punt p2;

public:
   Lijn(int x1, int y1, int x2, int y2) : p1(x1,y1),
```

```
p2(x2, y2)
   {
   }
   double lengte();
};

double Lijn::lengte()
{
   double dx, dy;

   dx = p1.haalx() - p2.haalx();
   dy = p1.haaly() - p2.haaly();
   return sqrt(dx*dx + dy*dy);
}

int main()
{
   Lijn ln(1,2,4,6);

   std::cout << ln.lengte() << std::endl;

   return 0;
}
```

We maken in het voorbeeld een klasse `Lijn`. Deze klasse bevat twee dataleden van het type `Punt`. Hiermee wordt het verband uitgedrukt dat een lijn twee punten verbindt. De `Punt` dataleden `p1` en `p2` zijn privaat. Dit betekent dat ze niet vrij toegankelijk zijn van buiten de klasse. De klasse `Lijn` kent één constructor. Deze constructor verwacht vier getallen als parameter. Dit zijn de twee coördinatenparen voor de begin- en eindpunten. Voor deze constructor is een speciale schrijfwijze toegepast. Als we de constructor als volgt zouden schrijven, dan zou de compiler een foutmelding geven:

```
Lijn::Lijn(int x1, int y1, int x2, int y2)
{
   p1.x = x1;
   p1.y = y1;
   p2.x = x2;
   p2.y = y2;
}
```

Wat is er nu fout aan deze schrijfwijze? De fout heeft te maken met de beveiliging van de private dataleden. De dataleden `x` en `y` van de twee `Punt` objecten zijn niet vrij toegankelijk. Vanuit de `Lijn` constructor is er alleen maar toegang tot de publieke klassefuncties van `p1` en `p2`. De dataleden `x` en `y` zijn privaat binnen de klasse `Punt` en daarom niet toegankelijk. Wel is de constructor van `Punt` toegankelijk. Er is echter in C++ geen mogelijkheid om rechtstreeks een constructor te starten als een functieoproep. Daarom kent C++ een speciale schrijfwijze om de dataleden van een klasse te initialiseren met een constructor. Daarom wordt de `Punt` constructor als volgt geschreven:

```
Lijn(int x1, int y1, int x2, int y2) : p1(x1,y1), p2(x2, y2)
{
}
```

Na de lijst van formele parameters volgt een dubbele punt. Hierna vermelden we de namen van de dataleden die binnen `Lijn` voorkomen. Dit zijn `p1` en `p2`. Na elk datalid noteren we de naam van de actuele parameters tussen haken. Zo wordt `p1` geïnitialiseerd met `x1` en `y1`; `p2` wordt geïnitialiseerd met `x2` en `y2`. Vanzelfsprekend moet er binnen de klasse `Punt` een constructor bestaan die met deze parameters overeen komt.

Met de functie lengte kan de lengte van een object van de klasse `Lijn` berekend worden.

Klassen afleiden

Als we van plan zijn om een bepaalde klasse uit te breiden met nieuwe dataleden of klassefuncties, dan zouden we rechtstreeks in de klassedefinitie deze dataleden of klassefuncties kunnen bijvoegen. Deze strategie heeft echter nadelen, zeker als de klasse reeds een tijd in gebruik is. Het is veiliger om de klasse ongewijzigd te laten en een afleiding te maken van deze klasse. Dit betekent dat we een nieuwe klasse ontwerpen die alle eigenschappen van een bestaande klasse overerft. Deze strategie heeft twee voordelen:

- de bestaande klasse hoeft niet gewijzigd te worden
- de functionaliteit van een bestaande klasse wordt volledig overgenomen in de nieuwe klasse
- de nieuwe klasse kan extra aangevuld worden met nieuwe dataleden en klassefuncties

Als we een klasse nodig hebben voor de voorstelling van een punt, waarbij ook nog in de klasse een naam opgeslagen wordt, dan is er een nieuwe klasse nodig. Als naam voor de nieuwe klasse kiezen we `PuntmetNaam`. De originele klasse `Punt` laten we ongewijzigd. We maken een afleiding van `Punt` en voegen er een naam aan toe.

De klasse definitie van `PuntmetNaam`ziet er als volgt uit:

```
#include <iostream>
#include "tekst.h"
#include "punt.h"

class PuntmetNaam : public Punt
{
private:
   Tekst naam;

public:
   PuntmetNaam(int ix, int iy, char *nm) : Punt(ix, iy), naam(nm)
   {
   }
   void druk();
};
```

Op dezelfde regel als de klassenaam schrijven we de naam van de klasse waarvan we willen afleiden. Het woord `public` geeft aan dat het gaat om een publieke afleiding. Dit betekent dat alle private dataleden en klassefuncties binnen `Punt` niet toegankelijk zijn vanuit de klassefuncties van `PuntmetNaam`. De protected leden van `Punt` zijn wel toegankelijk vanuit `PuntmetNaam` meer niet van buiten de klasse.

Binnen de klasse `PuntmetNaam` wordt een extra datalid bijgevoegd: `Tekst naam`. Hiermee kunnen we een naam opslaan. De constructor voor `PuntmetNaam` krijgt een extra parameter ten opzicht van die van `Punt`. De derde parameter is de string voor de naam. Door deze nieuwe constructor wordt de oude (overgeërfde) constructor niet meer toegankelijk. Dit is het herdefiniëren van een overgeërfde klassefunctie. De constructor bevat na de parameterlijst een dubbele punt en daarna een lijst van te initialiseren entiteiten: `Punt(ix, iy), naam(nm)`. Met de eerste notatie `Punt(ix, iy)` initialiseren worden de coördinaten `ix` en `iy` naar de `Punt` constructor doorgegeven. Met de tweede wordt de naam geïnitialiseerd. We zien dus twee soorten initialisators. Met een klassenaam geven we aan met welke gegevens de superklasse wordt geïnitialiseerd. Met een naam van een datalid geven we de initialisatie aan van een in de klasse zelf voorkomend klasselid.

```
void PuntmetNaam::druk()
{
   std::cout << "<" << naam.str();
   Punt::druk();
   std::cout << ">";
}
```

Net zoals de constructor wordt ook de functie `druk()` opnieuw gedefiniëerd. Ook hier is er een verwijzing naar een klassefunctie van de superklasse. Met `Punt::druk()` wordt een functie uit de superklasse

opgeroepen. De naam van de functie wordt voorafgegaan door de klassenaam en twee dubbele punten. Als we dit zouden weglaten, dan ontstaat er ongewild recursie.

In het hoofdprogramma worden twee objecten gedeclareerd. Telkens wordt `druk()` uitgevoerd.

```
int main()
{
   Punt p1(12,23);
   PuntmetNaam p2(56, 67, "oorsprong");

   p1.druk();
   std::cout << "\n";
   p2.druk();
   std::cout << "\n";

   return 0;
}
```

Het verband tussen de twee klassen kan grafisch weergegeven worden. Deze diagrammatechniek stamt uit *Universal Modelling Language*(UML).

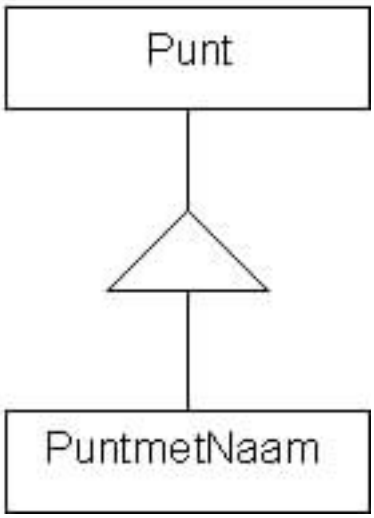

Figure 19: Erfenisvoorbeeld

Deze tekening geeft weer dat de klasse `PuntmetNaam` afgeleid is van de klasse `Punt`. Elke rechthoek stelt een klasse voor. De naam binnen de rechthoek stelt de klassenaam voor. Eventueel kunnen de dataleden en klassefuncties elk met een aparte rechthoek bijgevoegd worden.

Het diagramma ziet er dan als volgt uit:

In deze vorm toont het diagramma duidelijk dat door erfenis de klasse `PuntmetNaam` niet alleen `naam` als datalid heeft maar ook `x` en `y`.

Virtuele klassefuncties

Door het mechanisme van de afleiding is het mogelijk om een bepaalde klasse als basisklasse te gebruiken. Van deze basisklasse worden verschillende afleidingen gemaakt. De afgeleide klassen erven allemaal het gedrag van de basisklasse. De basisklasse bevat het gemeenschappelijk gedrag voor de verschillende afgeleide klassen. Dikwijls zijn er in de basisklasse klassefuncties nodig waarvan het gedrag pas definitief in de afgeleide klassen wordt bepaald. Daarom is het nodig dat de taal C++ voorzien is van een mechanisme om de keuze van welke klassefunctie gestart wordt (die uit de basisklasse of die uit de afgeleide klasse) te verschuiven tot bij de uitvoering van het programma. Dit mechanisme heet in C++ *virtuele functie*. In andere talen worden ook wel de termen *dynamische* of *late binding* gebruikt.

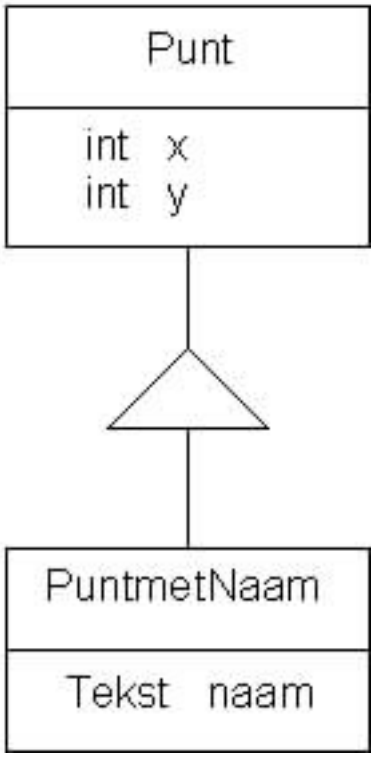

Figure 20: Erfenisvoorbeeld met klassevariabelen

Het concept virtuele functie is de kern van de objectgeoriënteerde kenmerken van de taal C++. Dit maakt het mogelijk om delen van software te ontwerpen die algemeen zijn en ook onafhankelijk zijn van alle later toe te voegen objecttypes.

Om dit concept duidelijk te maken starten we de uitleg van een voorbeeld. In dit voorbeeld maken we een algemene klasse die voor de opslag van een waarde wordt gebruikt. De virtuele functie maakt het mogelijk om specifiek gedrag in een afgeleide klasse te gebruiken vanuit een algemene klasse zonder de details te kennen van de afgeleide klassen en zonder afbreuk te doen aan de algemeenheid van de basisklasse. In het voorbeeld dat volgt, willen we gewoon een waarde op het scherm drukken zonder te weten van welk specifiek getaltype de waarde is.

Een abstracte klasse maken

We maken een basisklasse die gaat dienen voor de opslag van een waarde. Als klassenaam kiezen we de naam `Waarde`. De eerste letter is een hoofdletter, bijgevolg is dit een klassenaam. Deze klasse moet dienen om een waarde van een nog niet gekend type op te slaan. We wensen nu nog niet vast te leggen welk type gebruik zal worden want dan zou de klasse `Waarde` niet algemeen bruikbaar zijn. De klasse `Waarde` zou moeten kunnen werken met elk mogelijk getaltype.

```
class Waarde
{
private:
    // geen datalid

public:
   // hier plaatsen we de vrij toegankelijke klassefuncties
};
```

De beslissing om geen datalid voor de waarde in de klasse `Waarde` te plaatsen is voor dit voorbeeld een goede beslissing. We kunnen immers het datalid voor de waarde in de afgeleide klassen plaatsen. De klasse `Waarde` is bedoeld als basisklasse. We zullen van deze klasse nooit objecten maken. Zo komen we

meteen tot het begrip *abstracte klasse*. Een *abstracte klasse* is niet bedoeld om er concrete objecten mee te maken maar wel om een algemeen gedrag te bepalen voor een reeks afgeleide klassen.

Een virtuele functie maken

Bij dit voorbeeld is het gewenste algemeen gedrag van de klasse `Waarde` de mogelijkheid om de opgeslagen waarde op het scherm te drukken. Daarom plaatsen we een klassefunctie `druk()` in het publieke gedeelte.

```
class Waarde   // abstracte klasse
{
public:
   virtual void druk() = 0;
};
```

De schrijfwijze van het prototype van `druk()` vertoont twee nieuwe elementen:

- Voor `void` staat het woord `virtual`
- na de sluitende ronde haak staat `= 0`

Met het woord `virtual` geven we aan dat `druk()` een virtuele functie is. De precieze werking wordt later duidelijk. Na `druk()` staat er `= 0`. Hiermee geven we aan dat we voor `druk()` nog geen implementatie voorzien. Deze implementatie moet ingevuld worden in de verschillende afleidingen van de basisklasse. Deze `= 0` is niet nodig om de functie virtueel te maken maar wel om de klasse abstract te maken.

Van een abstracte klasse mogen we geen objecten maken. Wel is het mogelijk om een pointer of een referentie naar een abstracte klasse te maken.

```
Waarde w1;// FOUT
Waarde *pw1// GOED
```

Meestal zie je het gebruik van de abstracte klasse opduiken bij erfenis. Het is immers niet toegelaten om objecten te maken van een abstracte klasse; je moet afleidingen maken van de abstracte klasse en van deze afgeleide klassen kan je objecten maken. Het voordeel van een abstracte klasse is dat je een algemene klasse hebt die je kan gebruiken als je wilt verwijzen naar een aantal door erfenis verwante klassen. In een abstracte klasse kunnen implementaties van functies weggelaten worden met `= 0`. Je bent dan wel verplicht om in alle afgeleide klassen een implementatie te voorzien. Het is niet verplicht om een klasse abstract te maken maar dat betekent dan wel dat de niet-abstracte klasse een implementatie moet bevatten van al zijn functies.

Als we nagaan of het mogelijk is om de klasse `Waarde` niet abstract te maken, merk je dat we een implementatie voor `druk()` moeten voorzien. Maar er is geen datalid om een waarde bij te houden, dus kan je hoogstens een lege implementatie kunnen maken. Dit zou er dan zo uitzien:

```
class Waarde   // niet-abstracte klasse
{
public:
   virtual void druk()
   {
   }
};
```

De klasse is nu niet meer abstract en je kan er nu wel objecten van maken. Maar dit heeft niet veel zin omdat het objecten zijn die geen waarden opslaan. Het is daarom beter om de klasse abstract te houden.

Afleidingen maken

We maken twee afleidingen van de basisklasse. Een voor de opslag van een geheel getal en een voor de opslag van een reëel getal.

```
class IWaarde : public Waarde
{
private:
   int intwaarde;
```

```
public:
   IWaarde(int iw) : intwaarde(iw)
   {
   }
   virtual void druk();
};

class FWaarde : public Waarde
{
private:
   double floatwaarde;

public:
   FWaarde(double iw) : floatwaarde(iw)
   {
   }
   virtual void druk();
};
```

Elke van deze afleidingen krijgt een privaat datalid voor de opslag van de waarde. Het type van de waarde is telkens verschillend. In elke afgeleide klasse is er een constructor voorzien om het object te initialiseren. In elke afgeleide klasse wordt ook het prototype van `druk()` bijgevoegd. Dit betekent dat de functionaliteit van de functie `druk()` in de verschillende afgeleide klassen willen invullen. Het woord `virtual` wordt herhaald voor het prototype, dit is niet verplicht. Wel is het verplicht om bij het herdefiniëren van een virtuele functie in een afgeleide klasse dezelfde formele parameters te voorzien als in de basisklasse.

De twee `druk()` functies verschillen omdat de te drukken waarden van een ander type zijn:

```
void IWaarde::druk()
{
   std::cout << "geheel " << intwaarde;
}

void FWaarde::druk()
{
   std::cout << "reeel " << floatwaarde;
}
```

Objectcompatibiliteit

In programmeertalen zoals Pascal, Java, C en C++ is er een strikte typecontrole door de compiler. De omzetting van het ene type naar het andere type is niet altijd toegelaten. Op enkele uitzonderingen na is het verboden om verschillende types te gebruiken in toekenningen. In C en C++ kan deze beperking natuurlijk omzeild worden door de geforceerde omzetting (cast), maar deze programmeertechniek is niet elegant en veroorzaakt veel sneller fouten. In C++ wordt de cast-bewerking grotendeels overbodig door de mogelijkheid om in beperkte mate toch toekenningen te doen tussen verschillende types.

In C++ is het toegelaten om een toekenning te doen van pointers (dit geldt ook voor het referentietype) van een verschillende type. Er is wel één voorwaarde: de pointer aan de linkerzijde van de toekenning moet van een type zijn dat als superklasse voorkomt van de klasse van de pointer aan de rechterzijde van de toekenning. Deze uitzondering is de enige op de regel die zegt dat de twee types aan beide zijden van een toekenning gelijk moeten zijn.

Een voorbeeld maakt dit duidelijk:

```
Waarde *pw;
IWaarde *piw;
FWaarde *pfw;

pw = piw;   // ok, Waarde is de basisklasse van IWaarde
pw = pfw;   // ok, Waarde is de basisklasse van FWaarde
```

```
piw = pw;   // fout, Iwwaarde is geen basisklasse van Waarde
```

Deze compatibiliteit tussen verschillende pointertypes is nodig om gebruik te kunnen maken van de virtuele functies. Ter verduidelijking is hier nog het schema dat het verband tussen de verschillende klassen uit het voorbeeld weergeeft.

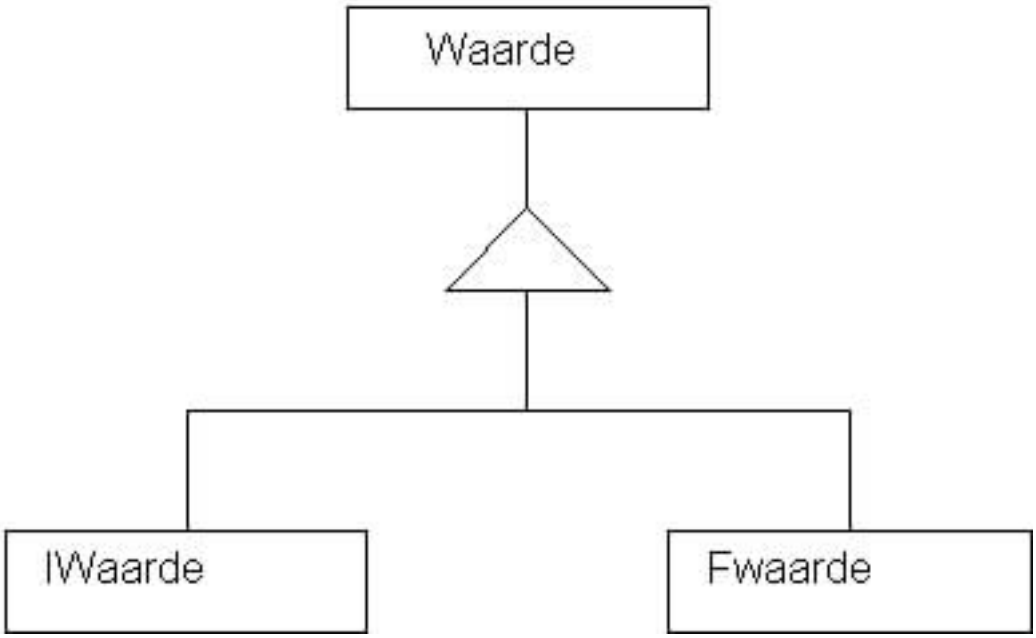

Figure 21: Twee klassen met een gemeenschappelijke superklasse

Het mechanisme van de virtuele functie

Als we een virtuele klassefunctie oproepen via een pointer naar de basisklasse komt het effect van de virtuele functie tot uiting.

```
IWaarde i1(5);
FWaarde f1(7.9);
Waarde *pw;

pw = &i1;
pw->druk();
pw = &f1;
pw->druk();
```

De pointer `pw` is een pointer naar een `Waarde` object. De declaratie en het gebruik van `pw` is toegelaten. Objecten van een abstracte klasse maken mag niet, pointers naar een abstracte klasse zijn wel toegelaten. Bij de eerste toekenning wijst `pw` naar een `IWaarde` object. Bij de oproep van `druk()` wordt de `IWaarde` variant van `druk()` gestart. Op het scherm verschijnt `5`. Bij de tweede toekenning wijst `pw` naar een `FWaarde` object. Bij de tweede oproep van `druk()` wordt de `FWaarde` variant van `druk()` gestart. Op het scherm verschijnt `7.9`. In elk object van een klasse met tenminste één virtuele functie zit extra informatie opgeslagen over de klasse van het object. Door deze informatie is het mogelijk dat bij de oproep van `druk()` bepaald wordt welke variant uit een van de afgeleide klassen wordt gestart.

Dit is de volledige tekst van het voorbeeld:

```
#include <iostream>

class Waarde   // abstracte klasse
{
public:
   virtual void druk() = 0;
```

```
};

class IWaarde : public Waarde
{
private:
   int intwaarde;

public:
   IWaarde(int iw) : intwaarde(iw)
   {
   }
   virtual void druk();
};

class FWaarde : public Waarde
{
private:
   double floatwaarde;

public:
   FWaarde(double iw) : floatwaarde(iw)
   {
   }
   virtual void druk();
};

void IWaarde::druk()
{
   std::cout << "geheel " << intwaarde;
}

void FWaarde::druk()
{
   std::cout << "reeel " << floatwaarde;
}

void toon(Waarde *pw)
{
   pw->druk();
}

int main()
{
   //Waarde ww;    fout Waarde is een abstracte klasse
   IWaarde i1(5);
   FWaarde f1(7.9);
   toon( &i1 );
   toon( &f1 );

   Waarde *pw1 = new IWaarde(256);
   Waarde *pw2 = new FWaarde(1.0/3.0);
   pw1->druk();
   pw2->druk();
}
```

In het voorbeeld wordt het principe van de virtuele functie tweemaal gedemonstreerd. Eenmaal worden adressen van objecten aan de functie `toon()` doorgegeven. De functie `toon` is ontworpen met het doel de waarde van het doorgegeven object te tonen. Het object wordt doorgegeven via zijn adres. Dit is efficiënter dan de volledige waarde door te geven. De functie `toon()` verwacht het adres van een `Waarde` object.

Vermits `Waarde` een abstracte basisklasse is van `IWaarde` of `FWaarde`, kan het doorgegeven object een `IWaarde` of een `FWaarde` object zijn. Vanuit de functie `toon()` bekeken kan niet op voorhand voorspeld worden of het doorgegeven object een `IWaarde` of een `FWaarde` is. Dit is de reden waarom we een virtuele functie gebruiken als mechanisme voor de start van `druk()`. Binnen `toon()` is er geen kennis nodig over de mogelijke specialisaties van `Waarde`. We kunnen dit voorbeeld besluiten met te zeggen dat een `Waarde` object toonbaar is; de waarde van het object kan verschillend zijn in de verschillende afleidingen. Dit verschil in gedrag wordt vastgelegd in de implementatie van de afgeleide klassen.

Omzettingen met `dynamic_cast`

In bepaalde gevallen willen we toch pointers van verschillende klassen kopiëren ook al is de kopieerrichting niet toegestaan. Je kan altijd een pointer naar een basisklasse kopiëren naar een pointer naar een basisklasse. De omgekeerde richting mag eigenlijk niet maar in het volgende voorbeeld doen we het toch.

In dit voorbeeld hebben we de klasse `A` als basisklasse en `B` en `C` zijn de afgeleide klassen. We maken een array van pointers naar `A` die gevuld wordt met de adressen van zowel `B` als `C` objecten. Deze kopiëerbewerking is toegestaan: van `B` of `C` naar `A` kopiëren mag omdat `B` en `C` afgeleid zijn van `A`. Zo wordt de array gemaakt:

```
A *tab[4] = { new B, new C, new C, new B };
```

Als we nu over de array lopen dan gebruiken we hiervoor uiteraard een pointer naar `A`, de basisklasse.

```
for (int i = 0; i < 4; i++)
{
   A *pa = tab[i];
   pa->doe();
}
```

We kunnen dan een methode van de klasse `A` starten: `pa->doe();`. Als we nu nog een extra methode willen starten die alleen maar in de `C` klasse voorkomt, zitten we vast. Dit gaat immers niet via een `A` pointer. De oplossing is dan `dynamic_cast`. Dit is een pseudofunctie die een omzetting doet van een pointer van een klasse naar een pointer van een andere klasse. De voorwaarde is wel dat de klassen gerelateerd moeten zijn met erfenis.

Dit schrijven we zo:

```
C *pc = dynamic_cast<C *>(pa);
```

Dit is natuurlijk een bewerking die alleen maar zin heeft als `pa` naar een `C` object wijst. Indien dat niet is, krijgen we een `nullptr` [31] als resultaat. Je kan dat testen met een `if`.

```
C *pc = dynamic_cast<C *>(pa);
if (pc != nullptr)
{
   pc->extra();
}
```

Met de pointer naar het `C` object kan je dan een specifieke methode starten. Hier is het volledige voorbeeld.

```
#include <iostream>

class A
{
public:
   virtual void doe() = 0;
};

class B: public A
{
public:
   void doe()
```

[31] Het is beter om niet meer `NULL` te gebruiken bij pointers, in C++11 is `nullptr` de nieuwe nulwaarde die je bij pointers kan gebruiken.

```
   {
      std::cout << "B::doe()" << std::endl;
   }
};

class C: public A
{
public:
   void doe()
   {
      std::cout << "C::doe()" << std::endl;
   }
   void extra()
   {
      std::cout << "C::extra()" << std::endl;
   }
};


int main(int argc, char *argv[])
{
   A *tab[4] = { new B, new C, new C, new B};

   for (int i = 0; i < 4; i++)
   {
      A *pa = tab[i];
      pa->doe();

      C *pc = dynamic_cast<C *>(pa);
      if (pc != nullptr)
      {
         pc->extra();
      }
   }
   return 0;
}
```

De `dynamic_cast` kan in bepaalde gevallen een uitkomst bieden, maar meestal kan je het probleem oplossen met enkel virtuele methoden. Merk op dat een `dynamic_cast` omzetting alleen maar werkt als de basisklasse tenminste één virtuele methode heeft.

Meervoudige erfenis

In het volgende voorbeeld erven de klassen `B` en `C` van de klasse `A`. Als dan de klasse `D` ook nog erft van zowel `B` als `C`, hebben we een probleem. Als je bij de methode `doe()` start bij een `D` object, weet de compiler niet of hij de `B.doe()` of de `C.doe()` moet starten. De klasse `D` heeft door meervoudige erfenis immers twee `doe()` met exact dezelfde naam. Hetzelfde probleem is er voor de datamember `waarde`. Deze komt tweemaal voor in de klasse `D`.

```
#include <iostream>

class A
{
public:
   int waarde;
   virtual void doe()
   {
   }
};
```

```
class B: public A
{
public:
   void doe()
   {
      std::cout << "B::doe()" << std::endl;
   }
};

class C: public A
{
public:
   void doe()
   {
      std::cout << "C::doe()" << std::endl;
   }
};

class D: public B, public C
{
public:
};

int main(int argc, char *argv[])
{
   D d;

   d.doe();
   return 0;
}
```

Een oplossing voor dit probleem is de klasse `D` een eigen methode `doe()` te geven die dan de andere overgeërfde `doe()` methoden start. Je moet dan expliciet de naam van de klasse vermelden bij de oproep van de methode. Hier is de uitbreiding van de klasse `D`.

```
class D: public B, public C
{
public:
   void doe()
   {
      B::doe();
      C::doe();
   }
};
```

In een aantal moderne programmeertalen heeft men meervoudige erfenis verboden omdat dit dikwijls de oorzaak is van onduidelijkheid in de erfenis. Meestal zal je in C++ de meervoudige erfenis ook niet nodig hebben.

Constante waarden en objecten

Constante objecten

Het is mogelijk om binnen de klassedeclaratie voorzieningen te treffen om constante objecten correct te behandelen. Een constant object is een object dat niet wijzigbaar is. Dit geven we aan bij de declaratie met het woord `const`. Bijvoorbeeld:

```
const Punt pc(45,67);
```

Het object `pc` is niet wijzigbaar; bijgevolg mogen voor dit object alleen klassefuncties gestart worden die garanderen dat er geen van de dataleden gewijzigd wordt. In de klassedeclaratie van `Punt` zien we na sommige functienamen het woord `const` staan. Hiermee wordt aangegeven dat de functie geen dataleden wijzigt. Indien er toch een wijziging van een datalid binnen een klassefunctie gebeurt, wordt dit als fout door de compiler gemeld.

```
#include <iostream>

// werken met constante objecten

class Punt
{
private:
   int x;
   int y;

public:
   Punt(int ix = 0, int iy = 0) : x(ix), y(iy)
   {
   }
   void druk() const
   {
      std::cout << "<" << x <<"," << y <<">" << std::endl;
   }

   // geen wijzigingen van dataleden toegelaten in const functies
   int haalx() const
   {
      // x++; fout
      return x;
   }
   int haaly() const
   {
      return y;
   }

   void zetx(int ix)
   {
      x = ix;
   }
   void zety(int iy)
   {
      y = iy;
   }
};

int main()
{
   Punt p1(2,3);
   const Punt p2(4,5);

   p1.druk();
   p2.druk();
   // p2.zetx(8); fout p2 kan niet gewijzigd worden

   return 0;
}
```

In `main` worden twee objecten gedeclareerd: `p1` en `p2`. De laatste is een constante. Dit betekent dat alleen

`const` functies bij dit object gestart kunnen worden.

Over het algemeen wordt aangeraden om in C++ zo weinig mogelijk gebruik te maken van `#define` om constanten te maken; we hebben immers de mogelijkheid om `const` te gebruiken. Je kan dus beter `const int grootte = 15;` schrijven in plaats van `#define GROOTTE 15`. Bij `const` is er een betere typecontrole dan bij `#define`. Een andere plek waar je `const` zinvol kan gebruiken, is de notatie van formele parameters. In het volgende prototype wordt een `Punt` als een referentie doorgegeven; dit is efficiënter dan als waardeparameter. En om te vermijden dat het `Punt` object gewijzigd wordt in de `toon()` methode hebben we er `const` voor geplaatst.

```
void toon(const Punt &pc);
```

Constante uitdrukkingen met `constexpr`

Nog uit te schrijven.

Statische leden in een klasse

Statische dataleden zijn leden waarvoor slechts éénmaal geheugenruimte wordt gereserveerd. In het volgende voorbeeld bestaat er binnen de klasse `Punt` een statisch dataveld aantal. De geheugenruimte bestaat slechts éénmaal. Elk object van het type `Punt` heeft een eigen `x` en `y` veld, maar het veld `aantal` is gemeenschappelijk voor de hele klasse. De toegang tot `aantal` verloopt niet via een object maar wel via de klasse. Het veld `aantal` wordt in dit voorbeeld gebruikt om bij te houden hoeveel objecten van de klasse `Punt` er bestaan. Deze boekhouding wordt met behulp van de constructor en destructor georganiseerd. Telkens als we de constructor of destructor doorlopen wordt aantal met 1 verhoogd of verlaagd. Door het feit dat `aantal` privaat is, zijn we er zeker van dat aantal niet buiten de klasse gewijzigd kan worden. Daarom zijn er ook toegangsfuncties bijgevoegd. Dit zijn `init_aantal()` en `haal_aantal()`. Dit zijn statische functies. Dit betekent dat ze niet in het kader van een object worden gestart, maar wel binnen de klasse.

De geheugenruimte voor een statisch datalid moet expliciet gereserveerd worden. Dit is anders dan in Java waar `static` dezelfde betekenis heeft.

```
#include <iostream>

// statische leden in een klasse

class Punt
{
private:
   int x;
   int y;
   static int aantal;

public:
   Punt(int ix = 0, int iy = 0) : x(ix), y(iy)
   {
      aantal++;
   }
   ~Punt()
   {
      aantal--;
   } // destructor

   void druk() const;
   int haalx() const
   {
      return(x);
   }
   int haaly() const
```

```
   {
      return(y);
   }
   void zetx(int ix)
   {
      x = ix;
   }
   void zety(int iy)
   {
      y = iy;
   }

   // statische klassefuncties worden zonder this opgeroepen
   static void init_aantal()
   {
      aantal = 0;
   }
   static int haal_aantal()
   {
      return( aantal);
   }
};

void druk();

void Punt::druk() const
{
   ::druk();// de twee dubbele punten zijn nodig voor recursie
            // te vermijden
   std::cout << "<" << x <<"," << y <<">" << std::endl;
}

void druk()
{
   std::cout << "cv10 ";
}

// de geheugenruimte voor statische dataleden
// moet expliciet gereserveerd worden

int Punt::aantal;

void fu()
{
   Punt p1(2,3);
   const Punt p2(4,5);

   p1.druk();
   p2.druk();

   std::cout << "aantal punten " << Punt::haal_aantal() << std::endl;
}

int main()
{
   // oproep zonder object
   Punt::init_aantal();
```

```
   fu();
   std::cout << "aantal punten " << Punt::haal_aantal() << std::endl;

   return 0;
}
```

Gebruik `static` om klasse-gerelateerde waarden te kunnen bijhouden. In dit voorbeeld is dit het aantal objecten dat van een bepaalde klassen aangemaakt wordt. Je kan ook `static` methoden in een klasse bijvoegen. Als je bijvoorbeeld met behulp van een methode nieuwe objecten van een klasse wil aanmaken, dan moet die methode `static` zijn.

Let op dat je niet teveel `static` gebruikt; dit is immers een verdoken vorm om globale variabelen te maken.

De [] operator bij reeksen

In dit voorbeeld wordt gedemonstreerd hoe het mogelijk is om de index bij arrays te controleren. Het is mogelijk om binnen een klasse een nieuwe betekenis te geven aan de `operator[]`. We kunnen zo een klasse laten werken als een array uit C. Dit geeft ons de mogelijkheid om de waarde van de index te controleren.

De klasse `Reeks` heeft een private pointer naar een array van `int`'s. Deze array wordt gereserveerd in de constructor en wordt terug vrijgegeven in de destructor. We houden binnen de klasse ook de lengte bij in het datalid `lengte`. De functie `controle_index()` wordt gebruikt om de index te controleren en eventueel een foutmelding te geven. Bij fout wordt het programma afgebroken met `exit()`. De `operator[]` ontvangt de index, controleert die en geeft dan de gewenste waarde terug. Het prototype ziet er als volgt uit:

```
int& operator [] ( unsigned long index )
```

Merk op dat het terugkeertype een referentietype is. Dit is nodig om de inhoud van de reeks te kunnen wijzigen.

In de methode `controle_index()` wordt er gebruik gemaakt van het preprocessor symbool `__LINE__`. Dit symbool wordt door de compiler vervangen door het regelnummer waar het symbool in de broncode staat.

```
#include <iostream>
#include <stdlib.h>

// een toepassing van de [] operator

class Reeks
{
private:
   int *data;
   unsigned long lengte;

protected:
   void controle_index(unsigned long index, int lijnnr )
   {
      if ( index >= lengte )
      {
         std::cout << "arrayindex-fout in regel "
              << lijnnr << " index " << index << std::endl;
         exit(1);
      }
   }

public:
   Reeks(unsigned long grootte)
```

```
   {
      lengte = grootte;// hou de grootte bij
      data = new int[grootte];// reserveer ruimte
      std::cout << "Reeks constructor\n";
   }
   ~Reeks()
   {
      delete [] data;// geef reeks vrij
      std::cout << "Reeks destructor\n";
   }

   int& operator[] ( unsigned long index )
   {
      controle_index( index, __LINE__ );// eerst controle
      return data[index];
   }
};

int main()
{
   Reeks lijst(10);

   for (int i=0; i<20; i++)
   {
      lijst[i] = i;
      std::cout << lijst[i];
   }

   return 0;
}
```

In `main()` wordt een reeks van 10 gehele getallen gecreëerd. De lijst wordt opgevuld met getallen en hier ontstaat een fout: bij `i = 10` wordt het programma afgebroken.

De oplossing die hier voorgesteld wordt, is niet volmaakt. In gebruik komt de klasse `Reeks` overeen met de ingebouwde arrays van C++. Je kan elk element bereiken met de rechte haken. Eén nadeel dat ook de ingebouwde arrays hebben, is dat de lengte van de reeks niet uitgebreid kan worden. Het enige voordeel is dat de index die gebruikt wordt om een element te bereiken, gecontroleerd wordt.

Het is ook zo dat de klasse `Reeks` alleen maar waarden van het type `int` kan bijhouden. Om te vermijden dat je `Reeks` moet herschrijven voor elk inhoudstype, is het beter om de klasse om te vormen tot een sjabloon.

Sjablonen

Sjablonen bij klassen: algemene reeksen

Het voorgaande voorbeeld was niet flexibel genoeg. Daarom wordt de klasse `Reeks` algemener gemaakt door sjablonen (*templates*) te gebruiken. Hierdoor is `Reeks` onafhankelijk van de opgeslagen soort. `Reeks` is niet meer een lijst van `int`'s maar wel van klasse `T` elementen. Dit zien we aan de naam van de klasse:

```
template<class T>
class Reeks
```

De klasse `Reeks` kent nu een typeparameter `T`. Hiermee kunnen we aangeven van welke klasse de gegevens zijn die in de reeks opgeslagen kunnen worden. In heel de klasse is `int` door `T` vervangen. De klasse `Reeks` is hierdoor algemener geworden. Voor het overige is `Reeks` niet gewijzigd.

```
#include <iostream>
#include <stdlib.h>
```

```
// reeks.h een algemene reeks

template<class T>
class Reeks
{
private:
   T *data;
   unsigned long lengte;

protected:
   void controle_index( unsigned long index, int lijnnr )
   {
      if ( index >= lengte )
      {
         std::cout << "arrayindex-fout in regel "
         << lijnnr<< " index " << index << std::endl;
         exit(1);
      }
   }

public:
   Reeks(unsigned long grootte)
   {
      lengte = grootte;
      data = new T[grootte];
      std::cout << "Reeks<T> constructor\n";
   }
   ~Reeks()
   {
      delete [] data;
      std::cout << "Reeks<T> destructor\n";
   }

   T& operator [] ( unsigned long index )
   {
      controle_index( index, __LINE__ );
      return data[index];
   }
};
```

In het hoofdprogramma maken we een reeks van `int` getallen. We moeten het type `int` als parameter meegeven:

```
Reeks<int>lijst(10);
Reeks<double>lijst2(100);
Reeks<Punt>lijst3(5);
```

Het actuele type wordt vermeld tussen kleiner en groter-dan tekens.

```
#include <iostream>
#include <stdlib.h>
#include "reeks.h"

// sjablonen: algemene reeksen

int main()
{
   Reeks<int> lijst(10);

   for (int i=0; i<20; i++)
```

```
   {
      lijst[i] = i;
      std::cout << lijst[i];
   }

   return 0;
}
```

De reeks in dit voorbeeld is zo algemeen geworden dat we dit type in veel gevallen kunnen toepassen. We hoeven niet meer voor elke soort gegevens een nieuwe reeks te ontwerpen. Voor elk type waarmee het sjabloon geparametreerd wordt, genereert de compiler een nieuwe reeks regels broncode waarin telkens `T` vervangen wordt door het gekozen concrete type.

Sjablonen bij functies

Je kan sjablonen (*templates*) ook toepassen bij functies. Hierdoor verkrijg je een functie die met een type geparametreerd is zodat de functie algemeen wordt. Hier is een voorbeeld van een functie die een doorgegeven variabele met één verhoogt. De functieparameter wordt als `T &w` geschreven. Je moet hier het referentietype gebruiken omdat de doorgegeven variabele gewijzigd moet kunnen worden. De variabele is van het type `T` en dat is de templateparameter.

```
template<class T>
void verhoog(T &w)
{
   w++;
}
```

Hier is het volledige voorbeeld.

```
#include <iostream>

template<class T>
void verhoog(T &w)
{
   w++;
}

class Waarde
{
private:
   int x;

public:
   Waarde(int ix) : x(ix)
   {
   }
   Waarde& operator++ ()     // prefix ++
   {
      x++;
      return *this;
   }

   // Maak de postfixoperator in functie van de prefixoperator.
   Waarde  operator++ (int)  // postfix ++
   {
      Waarde resultaat(*this);// maak een kopie voor het resultaat
      ++(*this);              // gebruik de prefixversie
      return resultaat;       // geef de kopie met de oude waarde terug
   }
```

```
   int get()
   {
      return x;
   }
};

int main()
{
   int    a = 7;
   double b = 3.5;
   Waarde c = 56;

   verhoog(a);
   verhoog(b);
   verhoog(c);

   std::cout << a << std::endl;
   std::cout << b << std::endl;
   std::cout << c.get() << std::endl;
}
```

Je kan de functie alleen maar gebruiken bij types die de postfix increment bewerking verstaan. Voor de types `int` en `double` is dat geen probleem maar als je bij de oproep van de `verhoog()` functie een object van een bepaalde klasse wil meegeven, dan moet die klasse wel de `operator++()` implementatie hebben.

```
Waarde& operator++()     // prefix ++
{
   x++;
   return *this;
}
```

Hierboven staat de prefixoperator. Deze heeft geen parameter. En hieronder staat de postfixoperator. Deze heeft wel een parameter maar die wordt niet gebruikt.

```
Waarde  operator++ (int)  // postfix ++
{
   Waarde resultaat(*this);// maak een kopie voor het resultaat
   ++(*this);              // gebruik de prefixversie
   return resultaat;       // geef de kopie met de oude waarde terug
}
```

Om de postfixoperator correct te laten werken, moet die de nog niet-geincrementeerde waarde teruggegeven. Daarvoor wordt de lokale variabele `resultaat` gebruikt. Met `++(*this)` wordt de prefixoperator gebruikt voor de verhoging.

Templatefuncties komen veelvuldig voor in de STL bibliotheek die deel uitmaakt van de C++ standaard.

Niet-klasse parameters bij templates

Soms is het handig om bij een templateklasse naast de klasseparameter ook een niet-klasseparameter te voorzien. In het volgende voorbeeld wordt een klasse `Rij` voorgesteld die intern een array van een zeker grootte bijhoudt. Naast het inhoudstype `T` wordt ook de lengte als een parameter `int lengte` doorgegeven. Deze parameter kan je dan binnen de klasse gewoon gebruiken.

```
#include <iostream>

template<class T, int lengte>
class Rij
{
private:
   T *tab;
```

```
public:
   Rij() : tab(new T[lengte])
   {
   }

   ~Rij()
   {
      delete [] tab;
   }
   T get(const int i)
   {
      return tab[i];
   }
};

int main()
{
   Rij<int, 7> a;
   Rij<bool, 20> b;
   Rij<double, 3> c;

}
```

Je ziet dat naast het inhoudstype telkens ook nog de grootte van de array moet doorgegeven worden. Er worden 3 arrays aangemaakt: één met 7 `int`'s, één met 20 `bool`'s en één met 3 `double`'s. Het voordeel van deze schrijfwijze is dat je geen extra paar ronde haken hoeft te gebruiken na de naam van de variabele om de parameter door te geven.

Uitzonderingen

Uitzonderingen (*exceptions*) laten toe om fouten op een gepaste manier af te handelen. In de voorgaande voorbeelden wordt de arrayfout drastisch afgehandeld. Het programma wordt gewoon afgebroken. Door gebruik te maken van uitzonderingen kan de foutafhandeling aangepast worden aan de noden van de gebruiker van een klasse. Binnen de klasse wordt in een foutsituatie de `throw` bewerking uitgevoerd. Hiermee wordt de fout aan de gebruiker van de klassen gemeld.

```
throw Fout();
```

Aan `throw` wordt een object van een klasse meegegeven. Hiermee wordt de fout geïdentificeerd. `Fout()` is de oproep van de standaard constructor zonder parameter. Om aan te geven dat we op mogelijke fouten reageren, definiëren we een `try`-blok.

```
try
{

}
catch(Fout)
{
   // foutafhandeling
}
```

Alle acties binnen het `try`-blok kunnen onderbroken worden door een `throw`. Alle bestaande objecten worden automatisch afgebroken door hun destructor. Dit geldt niet voor objecten die we dynamisch met `new` hebben gereserveerd. Het kan daarom nodig zijn om pointers naar een klasse op te nemen binnen een nieuwe klasse zodat de destructor van deze nieuwe klassen voor het vrijgeven van de dynamisch gecreëerde objecten zorgt.

```
#include <iostream>

// uitzonderingen
```

```
class Fout // dit is de foutklasse
{
};

class Gegeven
{
private:
   int getal;

public:
   Gegeven(int geg=0) : getal(geg)
   {
      std::cout << "Gegeven(" << getal<<") constructor" << std::endl;
   }
   ~Gegeven()
   {
      std::cout << "Gegeven(" << getal<<") destructor" << std::endl;
   }
};

void fu()
{
   Gegeven g = 2;
   Gegeven *pg;

   pg = new Gegeven(3);
   throw Fout();// g wordt afgebroken
   // pg niet
}

void fu2()
{
   Gegeven h = 4;

   fu();// h wordt afgebroken
}

int main()
{
   try// probeer fu2() te starten
   {
      fu2();
   }
   catch(Fout)// van fouten hier op
   {
      std::cout << "fout" << std::endl;
   }

   return 0;
}
```

In de functie `fu()` wordt een `throw` uitgevoerd. Hierdoor worden `h` en `g` automatisch afgebroken. Dit is niet het geval voor het object dat door `pg` wordt aangewezen. In `main()` komen we terecht in het `catch`-blok en wordt de nodige actie ondernomen.

Je moet de nodige maatregelen nemen zodat je van objecten die met `new` worden gemaakt toch verwijzingen bijhoudt. Hiermee kan je dan op een later tijdstip deze objecten vrijgeven. Een mogelijke oplossing is het bijhouden van de objecten in een lijst die buiten de `try/catch` constructie staat. Hierdoor blijft de lijst bestaan bij een `throw`. Dit soort constructies in een programma moet je altijd testen met `valgrind`.

Een oplossing voor de niet-vrijgave van geheugen dat met `new` gereserveerd is, is het gebruik van de STL type `unique_ptr` en `shared_ptr`. Deze types zijn nieuw in C++11 en garanderen de correcte vrijgave in alle omstandigheden. Hoe deze types ingezet kunnen worden, wordt later uitgelegd.

Een andere recente toevoeging in C++11 is het woord `noexcept`. Hiermee wordt aangegeven dat een methode gegarandeerd geen uitzondering zal gooien.

Een algemene reeks met uitzondering

De algemene reeks is nu met uitzonderingen beveiligd. De foutklasse `Arrayfout` houdt de foutmelding in tekstvorm bij. Binnen `controle_index()` kan een `throw` gestart worden.

```
// sjabloon Reeks met uitzondering
class Arrayfout
{
private:
   char *melding;

public:
   Arrayfout(char *p) : melding(p)
   {
   }
   char *foutmelding()
   {
      return( melding );
   }
};

template <class T>
class Reeks
{
private:
   T *data;
   unsigned long lengte;

protected:
   void controle_index( unsigned long index, int lijnnr )
   {
      if ( index >= lengte )
      {
         throw Arrayfout("arrayindex-fout");
      }
   }

public:
   Reeks(unsigned long grootte)
   {
      lengte = grootte;
      data = new T[grootte];
      std::cout << "Reeks constructor\n";
   }
   ~Reeks()
   {
      delete [] data;
      std::cout << "Reeks destructor\n";
   }
   T& operator [] ( unsigned long index )
   {
      controle_index( index, __LINE__ );
```

```
      return data[index];
   }
};
```

In het hoofdprogramma is een `try` en `catch`-blok bijgevoegd. Hiermee kunnen we de arrayfout opvangen.

```
#include <iostream>
#include <string.h>
#include "vreeks.h"

int main()
{
   try // probeer
   {
      Reeks<int> lijst(10); // een lijst van 10 int's

      for (int i=0; i<20; i++)
      {
         lijst[i] = i;
         std::cout << lijst[i];
      }
   }
   catch(Arrayfout &f) // vang arrayfout
   {
      std::cout << f.foutmelding(); // toon de foutmelding
   }

   return 0;
}
```

Het gebruik van uitzonderingen heeft als voordeel dat de ontwerper van hulpsoftware niet hoeft te kiezen voor één of andere drastische oplossing bij het optreden van een fout. Het enige wat er gebeurt, is het signaleren van de fout. Het is de verantwoordelijkheid van de gebruiker van de hulpsoftware om te bepalen wat er moet gebeuren als er een bepaalde fout optreedt.

De `Reeks` template met automatische uitbreiding

Het volgende voorbeeld is het bestand `nvreeks.h`. De hier voorgestelde templateklasse is in staat om de capaciteit van de opslag te vergroten zonder dat er gegevens verloren gaan. We zouden kunnen overwegen om de `Reeks` klasse in allerlei projecten te gaan gebruiken maar dat is geen goed idee. In de STL bibliotheek is er de `vector` klasse die veel beter geschikt is voor algemeen gebruik.

In het volgende testprogramma wordt het gebruik van deze reeks gedemonstreerd.

```
// cppvb17.cpp
#include <iostream>
#include "nvreeks.h"

int main()
{
   try
   {
      Reeks<int> tab(10);

      for (int i=0; i<20; i++)
      {
         int sq = i*i;
         tab.voegbij(sq);
      }
```

```
      for (int i=0; i<tab.grootte(); i++)
      {
         std::cout << i << " " << tab[i] << std::endl;
      }
   }
   catch(Arrayfout)
   {
      printf("Arrayfout\n");
   }
   catch(...)
   {
      std::cout << onbekende fout" << std::endl;
   }

   return 0;
}
```

In het bovenstaand programma zullen de `catch` blokken nooit bereikt worden omdat de reeks automatisch vergroot wordt. Er wordt gestart met grootte 10 en geëindigd met 20.

```
#ifndef _NVREEKS_H
#define _NVREEKS_H

class Arrayfout
{
};

template <class T>
class Reeks
{
private:
   T  *data;
   int lengte;  // gereserveerde grootte
   int aantal;  // aantal elementen in reeks

protected:
   void controle_index( int index, int lijnnr )
   {
      if ( index >= lengte )
      {
         throw Arrayfout();
         //std::cout << "arrayindex-fout";
         //exit(1);
      }
   }

public:
   Reeks(int grootte = 1) : aantal(0)
   {
      if (grootte <= 0)
      {
         grootte = 1;
      }
      lengte = grootte;
      data = new T[grootte];
   }
   ~Reeks()
   {
      delete [] data;
```

```
   }
   T& operator [] (int index )
   {
      controle_index( index, __LINE__ );
      return data[index];
   }
   void voegbij(T &element)
   {
      if (aantal >= lengte)
      {
         // reserveer nieuwe tabel en kopieer
         lengte *= 2; // dubbele lengte
         T *nw_data = new T[lengte];
         for (int i=0; i<aantal; i++)
         {
            nw_data[i] = data[i];
         }
         delete [] data;
         data = nw_data;
      }
      data[aantal++] = element;
   }
   int grootte()
   {
      return( aantal );
   }
};
#endif
```

Als datamembers wordt een array van het type `T`, de gereserveerde grootte (`lengte`) en het effectief aantal (`aantal`) ingevulde plaatsen in de array bijgehouden. Er worden dus 2 groottes bijgehouden: hoeveel plaats er is in het totaal en hoeveel van deze totale plaats al in gebruik is.

```
private:
   T  *data;    // de array
   int lengte;  // gereserveerde grootte
   int aantal;  // aantal elementen in reeks
```

De klasse `Reeks` heeft een constructor waaraan je een waarde groter dan nul moet doorgeven. Deze waarde stelt de grootte van de interne array voor. Indien je geen waarde doorgeeft, wordt 1 genomen als grootte. Indien toch een negatieve grootte wordt doorgegeven, dan wordt die vervangen door 1. Met deze uiteindelijke grootte wordt dan de array gereserveerd.

```
Reeks(int grootte = 1) : aantal(0)
{
   if (grootte <= 0)
   {
      grootte = 1;
   }
   lengte = grootte;
   data = new T[grootte];
}
```

De destructor doet niet meer dan de eerder gereserveerde array vrij te geven.

```
~Reeks()
{
   delete [] data;
}
```

Met de methode `operator[]` kan je zowel elementen ophalen als wijzigen. Je kan deze operator niet gebruiken om elementen toe te voegen.

```
   T& operator[] (int index )
   {
      controle_index( index, __LINE__ );
      return data[index];
   }
```

De methode `voegbij()` is verplicht om elementen toe te voegen in de reeks. Deze methode houdt de boekhouding van het aantal elementen bij. Indien er te weinig plaats is in de array, dan wordt een nieuwe array gereserveerd die dubbel zo groot is. De oude array wordt naar de nieuwe gekopieerd en de oude array wordt vrijgegeven. Op deze manier is er nooit plaats te kort.

```
void voegbij(T &element)
{
   if (aantal >= lengte)
   {
      // reserveer nieuwe tabel en kopieer
      lengte *= 2; // dubbele lengte
      T *nw_data = new T[lengte];

      // kopieer de oude naar de nieuwe array
      for (int i=0; i<aantal; i++)
      {
         nw_data[i] = data[i];
      }

      // verwijder de oude array
      delete [] data;

      // hou de nieuwe array bij
      data = nw_data;
   }
   data[aantal++] = element;
}
```

Dit voorbeeld laat zien hoe je zelf een algemeen inzetbare containerklasse kan maken. Het voorbeeld dient alleen maar om uit te leggen hoe sjablonen in C++ werken. Voor productiewerk kan je beter de containerklassen van STL gebruiken, meer bepaald `vector`.

Pointers opslaan in een container

In dit voorbeeld is voor het inhoudstype `T` gekozen voor de pointer `Info *`.

```
#include <stdio.h>
#include "nvreeks.h"

class Info
{
public:
   Info(int g): getal(g)
   {
   }
   void toon()
   {
      std::cout << getal << std::endl;
   }
private:
   int getal;
};

int main()
```

```
{
   try
   {
      Reeks<Info *> tab(10);


      for (int i=0; i<22; i++)
      {
         int sq = i*i;
         tab.voegbij(new Info(sq));
      }

      // alle Info's tonen
      for (int i=0; i<tab.grootte(); i++)
      {
         tab[i]->toon();
      }

      // alle Info's vrijgeven
      for (int i=0; i<tab.grootte(); i++)
      {
         delete tab[i];
      }
   }
   catch(Arrayfout)
   {
      std::cout << "Arrayfout\n";
   }
   catch(...)
   {
      std::cout << "onbekende fout\n";
   }
}
```

In dit voorbeeld zie je hoe een pointer als inhoudstype voor een containerklasse wordt gebruikt. Vermits de container de enige plek is waar de adressen van de `Info` objecten worden bijgehouden, moet je als laatste bewerking doorheen alle elementen van de container lopen om ze met `delete` vrij te geven.

Dit voorbeeld slaat de pointers naar de objecten in plaats van de objecten zelf, de declaratie van de lijst is dus `Reeks<Info *>` in plaats van `Reeks<Info>`. De reden is meervoudig:

- Door de objecten aan te wijzen met een pointer heb je een veel betere controle over de levensduur van de objecten.
- Er wordt vermeden dat objecten nodeloos gekopiëerd worden. Indien er meerdere kopieën van een object zouden ontstaan, kan het lastig worden om die allemaal in dezelfde toestand te houden.

Deze programmeerstijl met pointers naar objecten in C++ is semantisch dezelfde als de Java-stijl waar verwijzingen naar objecten in variabelen worden bijgehouden. Java kent geen pointers maar maar het gedrag van objectvariabelen in Java lijkt wel op dat van pointers.

Een container als klassevariabele

In dit voorbeeld is de container een datamember van een klasse geworden. Het voordeel hiervan is dat je in de destructor van de klasse waarin de container is opgenomen, alle elementen kan vrijgeven met `delete`.

```
#include <iostream>
#include "nvreeks.h"

class Info
```

```
{
public:
   Info(int g): getal(g)
   {
   }
   void toon()
   {
      std::cout <<  getal << std::endl;
   }

private:
   int getal;
};

class Gegevens
{
public:
   Gegevens(int n): lijst(n)
   {
      for (int i=0; i<n; i++)
      {
         lijst.voegbij(new Info(i * i));
      }
   }
   void toon()
   {
      std::cout << "Gegevens:" << std::endl;
      for (int i=0; i<lijst.grootte(); i++)
      {
         lijst[i]->toon();
      }
   }
   ~Gegevens()
   {
      for (int i=0; i<lijst.grootte(); i++)
      {
         delete lijst[i];
      }
   }

private:
   Reeks<Info *> lijst;
};


int main()
{
   try
   {
      Gegevens g(22);
      g.toon();
   }
   catch(Arrayfout)
   {
      printf("Arrayfout\n");
   }
   catch(...)
   {
```

```
      printf("onbekende fout\n");
   }

   return 0;
}
```

In de `main()` functie wordt `Gegevens g` als een lokale variabele gereserveerd. Wanneer we de try-blok buitengaan, bestaat `g` en wordt de container automatisch leeggemaakt. De destructor `Gegegevens::~Gegevens()` is hiervoor verantwoordelijk.

Hier doet `Reeks` dienst als 1-n relatie. Eén `Gegevens` object kan wijzen naar meerdere `Info` objecten.

Het `string` type

In C worden teksten voorgesteld door arrays van `char`. Het nadeel van deze oplossing is de grote kans op geheugenfouten. Een C array wordt altijd reserveerd met een vaste grootte. Zo bestaat `char t[8];` uit 8 bytes en de stringconstante `"abcdefg"` zal er juist in passen. Deze constante bestaat uit 7 zichtbare tekens en nog een onzichtbare binaire nul, die dient om het einde van de string te markeren. Nu is het zo dat door foute handelingen heel gemakkelijk voorbij het einde van een C array geschreven kan worden. Er is dan ook geen enkele controle op de grootte van de beschikbare ruimte in een C array. Om deze reden wordt afgeraden om in C++ de C-stijl strings te gebruiken. Je moet gebruik maken van de C++ `string`.

`string` is geen ingebouwd type maar een klasse die uitgerust is met voldoende operatoren zodat de klasse als een ingebouwd type kan gebruikt worden.

```
#include <string>

std::string s1 = "abc";
std::string s2("def");
```

Je kan een string ook in de uitvoer plaatsen.

```
std::cout << s1 << std::endl;
```

De loopnotatie van de `for` kan ook bij `string` gebruikt worden.

```
for (char t: s1)
{
   std::cout << t << std::endl;
}
```

Het kan ook op de conventionele manier.

```
for (unsigned int i=0; i<s1.size(); i++)
{
   std::cout << s1.at(i);
}
```

Of zoals bij arrays.

```
for (unsigned int i=0; i<s1.size(); i++)
{
   std::cout << s1[i];
}
```

De `size()` methode geeft de lengte van de string terug. En de `c_str()` geeft een pointer naar de interne string terug. Je mag die pointer alleen maar gebruiken voor leesbewerkingen en zeker niet voor schrijfbewerkingen of geheugenvrijgave.

```
const char *p = s1.c_str();
```

De `const` geeft aan dat alleen lezen toegestaan is.

Je kan string met een `+`-bewerking aan elkaar rijgen.

```
const char *p = s1.c_str();
```

Er zijn heel wat methoden binnen de string `klasse` maar de lijst is te lang om hier te behandelen.

STL (*Standard Template Library*) is een uitgebreide bibliotheek met containers en algoritmes die pas een tijdje na het ontstaan van C++ aan de taal is toegevoegd. STL is een deel van de standaard en elk platform waarvoor er een C++ compiler bestaat, zal dan ook STL ondersteunen. Door de evolutie van de taal C++ is STL een zeer uitgebreide bibliotheek geworden. Daarom kan niet elk aspect van STL in deze cursus behandeld worden. Alleen de belangrijkste concepten zullen toegelicht worden. Het eerste concept zijn de containers die bijna altijd de klassieke C arrays en gelinkte lijsten kunnen vervangen.

In dit deel wordt eerst gestart met een hoofdstuk dat in een aantal verschillende stappen uitlegt wat de alternatieven zijn om doorheen de gegevens van een lijst te lopen. De hoofdstukken erna leggen de concepten van containers en algoritmes uit.

Lijsten doorlopen

Voor we die containers bekijken, overlopen we een zestal C/C++ voorbeelden die telkens een alternatief tonen om doorheen de gegevens van een lijst te lopen. Het inzicht van hoe die alternatieven werken zal helpen om te begrijpen hoe STL achter de schermen werkt. Voor elk van deze alternatieven (behalve de laatste) wordt er standaard C of C++ gebruikt.

rechte haken [] bij een array

We starten met enkele eenvoudige voorbeelden die op verschillende wijzen een reeks of array doorlopen. Met deze voorbeelden zullen we beter zien waar de ontwerpers van STL hun inspiratie hebben gehaald. Het eerste voorbeeld is een C voorbeeld dat laat zien hoe je door een array loopt om elk element van de array te verdubbelen. Uiteraard gebruiken we dan een `for` voor de herhaling en zal de lusteller van de `for` dienst doen als index in de array. Het indexeren van een array is in C een tamelijk snelle bewerking[32] en daarom zal je deze programmeertechniek regelmatig tegenkomen.

```
void f1()
{
   int tab[] = { 6, 9, 4, 3, 1 };

   for (unsigned int i=0; i<sizeof(tab)/sizeof(int); i++)
   {
      tab[i] *= 2;
   }
}
```

In het bovenstaande voorbeeld wordt het aantal elementen in de array berekend door de lengte van de array in bytes te delen door de breedte van een `int` in bytes. Ook dit is een typische C techniek.

Met een pointer over een array lopen

In het volgende voorbeeld wordt een constante variabele gebruikt om het aantal elementen in de array vast te leggen. Nog een verschil is dat we een pointer `int *p` gebruiken om door de array te lopen. Hierdoor vervalt de indexering. Met `p++` wordt telkens naar het volgende element van de array verwezen. We maken dus gebruik van de alom bekende C pointerrekenkunde. De `for` gaat verder zolang het einde van de array niet bereikt is, dus zolang `p != tab + n` waar is. Opnieuw zien we pointerrekenkunde: `tab + n`. Hiermee wordt het adres van de eerste byte die geen deel meer uitmaakt van de array berekend.

```
void f2()
{
   const int n = 5;
   int tab[n] = { 6, 9, 4, 3, 1 };

   for (int *p = tab; p != tab + n; p++)
   {
```

[32] Vermits `tab[i]` kan geschreven worden als `*(tab+i)`, zie je dat er alleen maar een shift en een add instructie nodig zijn om te indexeren.

```
      *p *= 2;
   }
}
```

De bewerking uitvoeren via een pointer naar een functie

In versie 3 van het voorbeeld blijven we de pointer gebruiken als middel om alle elementen van de array te bereiken. Maar nu is de verdubbelbewerking in een aparte functie geplaatst. Het adres van deze functie wordt als parameter aan de functie `f3b` doorgegeven. De parameter is gedeclareerd als `void (*fu)(int &w))`. Hier staat dat `fu` een functie is die een referentie naar een `int` als parameter krijgt en die geen resultaat teruggeeft. De pointer naar een functie is een C techniek, de referentie in de parameter is een C++ schrijfwijze. Bij de oproep van `f3b` wordt de naam van de functie `verdubbel` als derde parameter meegegeven. In C is de naam van een functie het adres van de eerste instructie van die functie.

Het voordeel van een pointer naar een functie is dat je de herhaling algemeen houdt. De bewerking die op elk element toegepast moet worden, wordt van buitenaf doorgegeven.

```
void verdubbel(int &w)
{
   w *= 2;
}

void f3b(int *begin, int *end, void (*fu)(int &w))
{
   for (int *p = begin; p != end; p++)
   {
      (*fu)(*p);
   }
}

void f3()
{
   const int n = 5;
   int tab[n] = { 6, 9, 4, 3, 1 };

   f3b(tab, tab + n, verdubbel);
}
```

De functie `f3b()` krijgt voor de derde parameter de notatie `void (*fu)(int &w)`. Dit is een pointer naar een functie die een referentie naar een `int` krijgt en geen resultaat teruggeeft. Via deze pointer wordt de functie opgeroepen. Omdat het hier over een pointer gaat, moet je een `*` voor de pointernaam schrijven en dat geheel wordt dan tussen haken geplaatst. Daarna volgen de ronde haken voor de functieoproep en hiertussen wordt de actuele parameter geplaatst. Dat is in dit geval `*p`. Hier is ook een `*` nodig omdat we niet de pointer maar wel de aangewezen `int` willen laten veranderen. Dit is dan uiteindelijk de notatie van de oproep:

```
(*fu)(*p);
```

De bewerking uitvoeren via een pointer naar een templatefunctie

Hier volgt versie 4. Overal waar `int` voorkomt als type van de te verdubbelen waarde, is die naam vervangen door `T`. De betrokken functies zijn nu templatefuncties geworden. Uiteraard kan je het voorbeeld enkel compileren met een C++ compiler. De `for` herhaling is nu onafhankelijk van het type van de waarden in de array.

```
template<class T>
void verdubb(T &w)
{
   w *= 2;
}
```

```
template<class T>
void f4b(T *begin, T *end, void (*fu)(T &w))
{
   for (T *p = begin; p != end; p++)
   {
      (*fu)(*p);
   }
}

void f4()
{
   const int n = 5;
   int tab[n] = { 6, 9, 4, 3, 1 };

   f4b(tab, tab + n, verdubb);
}
```

De bewerking uitvoeren via een functieobject

In versie 5 wordt verdubbelfunctie vervangen door een functieobject. Wat is een functieobject? Dit is object van een klasse waarin naast eventueel een constructor enkel de operator voor de ronde haken `T &operator()(T &w)` voorkomt. In dit geval maken we de klasse `Verdubbel` waarin deze operator voorkomt. Bij de oproep van de functie `f5b` wordt als derde parameter een statisch gereserveerd `Verdubbel` object meegegeven. Dit wordt geschreven als `Verdubbel<int>()`. De formele parameter wordt als `A actie` geschreven. De oproep van de operator () wordt dan als `actie(*p)` geschreven. Dat ziet er uiteraard uit als de oproep van een functie of methode. Een functieobject is de C++ equivalentie van de C functiepointer.

```
template<class T>
class Verdubbel
{
public:
   T &operator()(T &w)
   {
      w *= 2;
      return w;
   }
};

template<class T, class A>
void f5b(T *begin, T *end, A actie)
{
   for (T *p = begin; p != end; p++)
   {
      actie(*p);
      printf("%d\n", *p);
   }
}

void f5()
{
   const int n = 5;
   int tab[n] = { 6, 9, 4, 3, 1 };
   f5b(tab, tab + n, Verdubbel<int>());
}
```

De bewerking uitvoeren via een pointer naar een lambdafunctie

Omdat het nogal omslachtig is om een volledig nieuwe klasse te moeten ontwerpen enkel voor het vastleggen van een bewerking, heeft de C++ standaardisatiecommissie gezocht naar een andere en kortere notatie.

En zo zijn ook de zogenaamde *Lambda functies* in C++ aanbeland. Een lambdafunctie is een soort veredelde pointer naar een functie. Je kan hiermee een anonieme functie maken die je in een soort pointer kan bijhouden en die je via parameter kan doorgeven aan andere functies en methoden. De parameter om een lambdafunctie op te vangen is opnieuw een pointer naar een functie: `void (*actie)(int &)`. De definitie van de lambdafunctie is redelijk compact: `[](int &w) { w *= 2; }`. Hiermee wordt een functie vastgelegd die een referentie naar een `int` verwacht en deze `int` waarde verdubbeld. Je moet een referentie gebruiken omdat de waarde gewijzigd wordt. Je zou ook een pointer naar een `int` kunnen gebruiken in plaats van de referentie.

Je kan dit voorbeeld enkel met een recente compiler compileren die de C++11 standaard ondersteunt. Lambdafunctie zijn een recente toevoeging aan C++. Dit concept stamt uit de functionele programmeertalen.

```
template<class T>
void f6b(T *begin, T *end, void (*actie)(int &))
{
   for (T *p = begin; p != end; p++)
   {
      actie(*p);
      printf("%d\n", *p);
   }
}

void f6()
{
   const int n = 5;
   int tab[n] = { 6, 9, 4, 3, 1 };
   f6b(tab, tab + n, [](int &w) { w *= 2; });
}
```

Uiteindelijk hebben we de hierboven staande zesde versie van het voorbeeld. Met dit voorbeeld heb je nu een idee hoe de algoritmes, die veelvuldig in de STL bibliotheek voorkomen, geïmplementeerd worden. Het algoritme dat in dit voorbeeld gedemonstreerd wordt, is het uitvoeren van een bewerking op elk element van een reeks. Zowel het type van de elementen (template) en de bewerking (lambdafunctie) kunnen in het algoritme geparametreerd worden.

Een `vector` voorbeeld

De eerste containerklasse van STL die aan bod komt is `vector`. Deze klasse maakt intern een lijst aan van allemaal elementen die aaneengesloten in het geheugen geplaatst worden. De onderliggende technologie is de C array. Je kan de rechte haken `[]` gebruiken om de elementen te bereiken. De `vector` houdt zelf bij hoeveel elementen er aan de lijst zijn toegevoegd en indien nodig, wordt de interne array automatisch verlengd indien er plaats te kort is.

In het volgende voorbeeld wordt het gebruik van `vector` getoond.

```
#include <vector>

class Punt
{
private:
   int x;
   int y;

public:
   Punt(int ix, int iy) : x(ix), y(iy)
   {
   }

   void toon()
```

```
   {
      std::cout << x << ", " << y << std::endl;
   }
};


int main()
{
   vector<Punt *> lijst;

   for (int i=0; i<20; i++)
   {
      Punt *p = new Punt(1+i,2+i);
      lijst.push_back(p);
   }

   for (int i= 0;i<lijst.size(); i++)
   {
      lijst[i]->toon();
   }

   vector<Punt *>::iterator it = lijst.begin();
   while (it != lijst.end())
   {
     Punt *q = *it;
     q->toon();

     it++;
   }


   for (int i= 0;i<lijst.size(); i++)
   {
      delete lijst[i];
   }

   return 0;
}
```

In het voorbeeld wordt deze lijst gedeclareerd: `vector<Punt *> lijst;`. Dit betekent dat we pointers naar `Punt` objecten opslaan. Je kan elementen bijvoegen met de `push_back()` methode. Dit is bij `vector` de snelste methode om elementen bij te voegen in de lijst. De grootte van de lijst kan je opvragen met `size()`.

Er is ook een alternatieve manier om doorheen de elementen van de lijst te lopen. Hierbij wordt gebruik gemaakt van een iterator.

```
vector<Punt *>::iterator it = lijst.begin();
while (it != lijst.end())
{
   Punt *q = *it;
   q->toon();

   it++;
}
```

Een iterator is een soort gegeneraliseerde pointer waarmee het mogelijk is om door de lijst te lopen. Een iterator kan je op dezelfde wijze gebruiken als een pointer ook al is een iterator geen pointer. Zo geven `lijst.begin()` en `lijst.begin()` respectievelijk het begin en het einde van de lijst. Met de `++` ga je naar het volgende element en met de `*` kan je de waarde ophalen die op dat moment door de

iterator aangewezen wordt. Merk op dat in het bovenstaande fragment `it` van het type `vector<Punt *>::iterator` is en dat `*it` van het type `Punt *` is.

In de notatie `*it` is de `*` in feite de oproep van de `operator*()`. Door deze notatie lijkt het erop dat `it` een pointer is, maar dat is het niet. `it` is een object van de klasse `vector<Punt *>::iterator`. Men heeft er wel voor gekozen om iterators op pointers te laten lijken. Hierdoor kan je de vertrouwde ster gebruiken bij iterators net zoals bij pointers.

Vanaf C++11 kan je door de lijst lopen met een nieuwe schrijfwijze bij de `for`. Deze schrijfwijze is duidelijk geïnspireerd door Java.

```
vector<int> reeks;
reeks.push_back(5);
reeks.push_back(6);
reeks.push_back(7);

for (int i: reeks)
{
   std::cout << i << std::endl;
}

for (int &i: reeks)
{
   i *= 2;
}

for (auto i: reeks)
{
   std::cout << i << std::endl;
}
```

De eerste `for` is exact hetzelfde als de schrijfwijze in Java. Bij de tweede `for` wordt het referentietype gebruikt; hierdoor kan je de verkregen waarde wijzigen. En in het derde voorbeeld kan je het type vervangen door `auto`. Het gebruik van `auto` als type is nieuw sinds C++11. In dit geval zeg je hiermee dat de variabele `i` als type het inhoudstype van `reeks` moet krijgen; dat wordt in dit concrete voorbeeld het type `int`. Het gebruik van `auto` bespaart het schrijfwerk van lange typeuitdrukkingen.

Een kortere manier om een `vector` te vullen is deze:

```
vector<int> tab = { 3, 5, 7, 8, 9};
```

Dit is dezelfde schrijfwijze zoals bij de arrays in C.

Tot slot is er nog het volgende fragment dat toont hoe je pointerrekenkunde kan toepassen bij iterators. Je mag een `int` waarde optellen bij een iterator. Het voorbeeld verwijdert het derde tot en met het laatste element.

```
lijst.erase(lijst.begin() + 2, lijst.end());
```

`lijst.begin()` geeft als resultaat een iterator terug die wijst naar het eerste element van de lijst. Door er 2 bij op te tellen zal de iterator wijzen naar het derde element. Vanaf daar worden de lementen uit de lijst verwijderd.

Indien je regelmatig elementen uit een `vector` wilt verwijderen, dan is het wellicht beter om de `vector` te vervangen door `deque`. Deze container heeft gelijkaardige eigenschappen zoals `vector` maar het verwijderen van elementen is er wel sneller.

De STL `list`

Een `list` voorbeeld

Hier is het voorgaande voorbeeld overgenomen en `vector` door `list` vervangen. De herhaling waarbij de rechte haken `[]` worden gebruikt voor indexering, is geschrapt. Bij `list` en andere containers is dit niet mogelijk.

```
#include <iostream>
#include <list>

class Punt
{
private:
   int x;
   int y;

public:
   Punt(int ix, int iy) : x(ix), y(iy)
   {
   }
   ~Punt()
   {
      std::cout << "~Punt" << std::endl;
   }

   void toon()
   {
      std::cout << x << ", " << y << std::endl;
   }
};


int main()
{
   list<Punt *> lijst;

   for (int i=0; i<20; i++)
   {
      Punt *p = new Punt(1+i,2+i);
      lijst.push_back(p);
   }

   list<Punt *>::iterator it = lijst.begin();
   while (it != lijst.end())
   {
      Punt *q = *it;
      q->toon();

      it++;
   }

   it = lijst.begin();
   while (it != lijst.end())
   {
      Punt *q = *it;
      delete q;

      it++;
   }

   return 0;
}
```

De achterliggende implementatie bij `list` is de dubbelgelinkte lijst. Vermits hier de rechte haken niet meer kunnen gebruikt worden voor het doorlopen van de lijst, moet je gebruik maken van een iterator.

Merk ook op dat zowel in het `vector` als in het `list` de `Punt` objecten dynamisch zijn gereserveerd. Dit betekent dat de objecten op een zeker moment vrijgegeven zullen moeten worden. De STL container neemt dit niet voor zijn rekening. Je zal zelf nog een laatste keer door de lijst moeten lopen om de `Punt` één voor één vrij te geven.

Geheugenlekken opsporen

In dit voorbeeld wordt een ingenieus systeem toegepast om geheugenlekken op te sporen. Hiervoor moet wel een opgelegd ontwerppatroon gevolgd worden. De hier voorgestelde techniek is ooit ontworpen om geheugenlekken op te sporen in *pasrun*. Dit is een implementatie van een Virtuele Pascal Machine. Er werden met succes een aantal lekken opgespoord en gedicht. Ondertussen is er de tool `valgrind` waarmee je hetzelfde kan bereiken en hoef je deze techniek niet meer te gebruiken. Om didactische redenen is het complete voorbeeld van deze lekdetectietechniek toch in deze cursustekst gebleven. Het voorbeeld toont hoe je een aantal C++ technieken kan gebruiken. Wie echter geen tijd heeft, mag het voorbeeld overslaan.

De opgelegde ontwerptechniek is de volgende: elke klasse in je ontwerp moet afgeleid worden van de klasse `GElem`. Deze erfenis is verplicht, anders wordt er geen lekcontrole gedaan op de betrokken klasse.

Er moet ook één enkel globaal `GLijst` object gemaakt worden. Op het einde van het programma wordt dan automatisch gemeld welke objecten nog niet vrijgegeven worden.

Het hele voorbeeld is opgeslagen in één enkel headerbestand.

```
#ifndef DBG_GEHEUGEN

#include <iostream>
#include <list>

// Deze klasse is een element bedoeld om in een lijst
// opgeslagen te worden. Omdat vanuit deze klasse verwezen
// wordt naar de lijstklasse en omdat dit compilatiefouten
// geeft, wordt de lijstklasse als parameter class T doorgegeven

template<class T>
class GElem_tm     // dit is de elementklasse met template parameter
{
public:
   GElem_tm()
   {
      std::cout << "GElem()" << std::endl;
      T::haal().voegbij(this); // T wordt GLijst
   }
   ~GElem_tm()
   {
      std::cout << "~GElem()" << std::endl;
      T::haal().verwijder(this); // T wordt GLijst
   }

   // deze functie moet een andere invulling
   // krijgen in de afleiding
   virtual void toon()
   {
      std::cout << "GElem" << std::endl;
   }
};

class GLijst;  // voorwaartse declaratie van een klasse
               // hiermee wordt de compiler verteld dat
               // deze naam een klasse is
```

```
          // dit is de elementklasse met template parameter
          // de actuele parameter is GLijst
typedef GElem_tm<GLijst> GElem;

// De klasse GLijst houd een lijst van GElem bij
// elke in de toepassing gebruikte klasse
// moet overerven van GElem.

class GLijst
{
public:
   GLijst()
   {
      std::cout << "GLijst()" << std::endl;
   }
   void voegbij(GElem *em)
   {
      lijst.push_back(em);
   }
   void verwijder(GElem *em)
   {
      lijst.remove(em);
   }
   void toon()
   {
      std::cout << "GLijst:" << std::endl;
      std::cout << "lijst van nog niet vrijgegeven objecten" << std::endl;

      list<GElem *>::iterator it = lijst.begin();
      while (it != lijst.end())
      {
         GElem *q = *it;
         q->toon();

         it++;
      }

   }
   ~GLijst()
   {
      std::cout << "~GLijst" << std::endl;
      list<GElem *>::iterator it = lijst.begin();
      while (it != lijst.end())
      {
         printf("element vergeten vrij te geven\n");
         GElem *q = *it;
         q->toon();

      /* Deze delete zorgt er ook voor dat het GElem uit
         de lijst verwijderd is. Om problemen te vermijden
         met it laten we it steeds naar het begin van de lijst
         wijzen.
       */
         // verwijder GElem uit lijst en geheugen
         delete *it;

         // het begin van de lijst is veranderd,
         // daarom opnieuw instellen
```

```
         it = lijst.begin();

         // dit zou fout zijn
         //  it++;
      }
   }

   // deze functie geeft een referentie naar het
   // ene en enige GLijst object terug
   static GLijst &haal()
   {
       return geheugen;
   }
   // dit is een statisch GLijst object
   static GLijst geheugen;

private:
   list<GElem *> lijst;
};

#endif
```

Nu wordt het headerbestand (gedeeltelijk) uitgelegd. Eerst wordt de klasse `GElem_tm` vastgelegd. Deze klasse stelt een element voor in de lijst van gereserveerde objecten. Dit is een template klasse waarbij `T` nog vastgelegd wordt.

```
template<class T>
class GElem_tm     // dit is de elementklasse met template parameter
{
};
```

Dan volgt er een voorwaartse declaratie. Hiermee wordt aangegeven dat de klasse `GLijst` later nog volgt. Hierdoor is de naam van deze klasse al bekend gemaakt aan de compiler.

```
class GLijst;

typedef GElem_tm<GLijst> GElem;
```

Met een `typedef` wordt de klasse `GElem` als een nieuw type vastgelegd. Hierbij wordt de klasse `GElem_tm` geparametreerd met de klasse `GLijst`. Deze omslachtige techniek waarbij een klasse in twee stappen wordt gemaakt, is noodzakelijk om compilatiefouten te vermijden.

Daarna volgt de definitie van de klasse `GLijst`.

```
class GLijst
{
};
```

En hier is een voorbeeldprogramma waarin geheugenlekkentest wordt toegepast.

```
#include <iostream>
#include "dbg-geheugen.h"

// dit is een klasse die getest wordt
// tegen geheugenlekken, vandaar
// de erfenis van GElem

class Punt : public GElem
{
private:
   int x;
   int y;
```

```
public:
   Punt(int ix, int iy) : x(ix), y(iy)
   {
   }
   ~Punt()
   {
      std::cout << "~Punt" << std::endl;
   }

   void toon()
   {
      std::cout << "Punt: " << x << ", ", << y << std::endl;
   }
};


// dit is het object met daarin de lijst
// de constructor en destructor worden
// automatisch gestart
GLijst GLijst::geheugen;

int main()
{
   std::cout << "start" << std::endl;

   Punt *p1 = new Punt(5,6);
   Punt *p2 = new Punt(7,8);
   Punt *p3 = new Punt(10,10);


   // deze toon geeft al aan dat er
   // Punt objecten in het geheugen
   // zitten
   GLijst::geheugen.toon();
   std::cout << "einde" << std::endl;

   // nadat deze accolade overschreden is,
   // wordt de destructor van GLijst gestart
   // en die meldt dat er nog 3 objecten
   // in het geheugen zitten

   return 0;
}
```

In de bovenstaande `main()` is heel de toepassing opgenomen. Vooraf wordt de klasse `Punt` vastgelegd die erft van `Gelem`.

Verder moet je de static (en bijgevolg globale) variabele voor de lijst declareren.

```
GLijst GLijst::geheugen;
```

Automatisch zal eerst de constructor en later de destructor van de variabele `geheugen` opgeroepen worden. Het is de laatste die een lijst van niet-vrijgegeven objecten op het scherm zal afdrukken.

De enige wijziging aan een programma waarin deze techniek wordt toegepast, is het bijvoegen van erfenis bij elke in de test betrokken klasse. Vermits je in C++ meervoudige erfenis kan toepassen, is dit voor elke eigen gemaakte klasse haalbaar. Ook is dit voorbeeld didactisch bedoeld; voor echte testen is `valgrind` het aangewezen hulpmiddel.

De `set` container

De `set` container slaat unieke waarden op. Dit betekent dat een welbepaalde waarde slechts éénmaal kan voorkomen in de verzameling. De verzameling wordt altijd gesorteerd bijgehouden. Dit komt door de onderliggende implementatie. Dit is een gebalanceerde binaire boom. Dit type implementatie is ideaal om gegevens op een gesorteerde wijze bij te houden. Het balanceren zorgt ervoor dat het zoeken snel verloopt. En dat zoeken is nodig om te achterhalen of een element al in de lijst voorkomt.

```
#include <iostream>
#include <set>

set<int> getallen;

getallen.insert(6);
getallen.insert(1);
getallen.insert(7);
getallen.insert(4);
getallen.insert(8);

getallen.erase(7);

for (auto it=getallen.begin(); it != getallen.end(); it++)
{
   std::cout << *it << std::endl;
}
```

Het bovenstaande voorbeeld toont de getallen in stijgende volgorde.

Voor het sorteren wordt er verwacht dat het gebruikte type de kleiner dan vergelijking verstaat. Voor een `int` is dat evident. Indien nodig, moet je zelf een vergelijking bijvoegen. In het volgende voorbeeld is dat met een klasse voor een functieobject opgelost. Hierdoor wordt de volgorde van de getallen in de `set` omgekeerd.

```
class Comp
{
public:
   bool operator()(const int &a, const int &b)
   {
      return a > b;
   }
};

void f()
{
   set<int, Comp> v;
   v.insert(6);
   v.insert(1);
   v.insert(8);
   v.insert(3);
   v.insert(2);

   for (auto it=v.begin(); it != v.end(); it++)
   {
      std::cout << *it << std::endl;
   }
}
```

Hierboven zie je de klasse `Comp` die dienst doet als vergelijker. Met deze klasse wordt de `set` geparametreerd. Deze versie van het voorbeeld toont de getallen in dalende volgorde.

Met de komst van de lambdafuncties in C++11 kan je hetzelfde ook oplossen met een lambdafunctie. Dat

wordt in het volgende fragment gedemonstreerd.

```
auto cmp = [](const int &a, const int &b)->bool { return a > b; };
set<int, decltype(cmp)> v(cmp);
```

De schrijfwijze is misschien wel omslachtig, maar je moet geen extra klasse definiëren. `cmp` is een verwijzing naar een lambdafunctie die voor de omgekeerde volgorde zorgt. Het type van `cmp` willen we niet weten, vandaar dat er `auto` staat. Met de ingebouwde `decltype` operator haal je het type van een uitdrukking. In dit geval is dit nodig om de `set` te parametreren. Merk op dat de variabele `cmp` tweemaal in de declaratie van de `set` voorkomt: éénmaal als parameter van `decltype()` en éénmaal als parameter voor de constructor. Met `decltype()` haal je het type van een uitdrukking. Samengevat geeft de bovenstaande declaratie een `set` met naam `v` die de getallen in dalende volgorde zal bijhouden.

De `map` container

Met een `map` kan je een verzameling van key/value paren maken. Elk paar wordt in de map opgeslagen als een `pair` object. `pair` is een klasse waarmee je associaties tussen twee waarden kan maken. In het volgende voorbeeld zijn de paren van de klasse `pair<string, int>`. Deze `pair` heeft twee datamembers: `first` en `second`. Hiermee kan je de twee waarden ophalen.

```
#include <map>

map<string, int> freq;
freq["maandag"]   = 4;
freq["dinsdag"]   = 3;
freq["woensdag"]  = 4;
freq["donderdag"] = 5;
freq["vrijdag"]   = 6;

std::cout << freq["woensdag"] << std::endl;

for (auto it=freq.begin(); it != freq.end(); it++)
{
   std::cout << (*it).first << ":" << (*it).second << std::endl;
}
```

Je gebruikt de rechte haken zowel om voor een bepaalde sleutel een waarde bij te voegen en om met het behulp van een sleutel een waarde op te halen. Je kan met een iterator doorheen de verzameling lopen en telkens de sleutel op te halen met `.first` en de waarde met `.second`.

Bij een `map` kan je slechts één waarde voor een bepaalde sleutel opslaan. Als je alle paren van de `map` overloopt, gebeurt dat in stijgende volgorde van de sleutel (de eerste waarde).

Algoritmes

Er zijn verschillende soorten algoritmes in STL. Het enige die hier besproken wordt, is `sort`. De meeste algoritmes zijn als een templatefunctie geschreven.

```
#include <vector>
#include <algorithm>

vector<int> l;
l.push_back(4);
l.push_back(2);
l.push_back(7);
l.push_back(1);

sort(l.begin(), l.end());

auto i = l.begin();
```

```
while (i != l.end())
{
   std::cout << *i << std::endl;

   i++;
}
```

In het bovenstaande voorbeeld wordt een `vector` gevuld en daarna gesorteerd. Bij de oproep van `sort()` moet je twee iterators meegeven: `l.begin()` en `l.end()`. Deze iterators geven het begin en einde van de lijst aan.

Je kan ook de `sort()` functie ook gebruiken om een gewone array te sorteren. In dat geval geven we pointers mee om het begin en einde van de lijst aan te geven.

```
int tab[] = {8, 2, 9, 4, 5, 6, 2, 7};

sort(tab, tab+sizeof(tab)/sizeof(int));

for (int *p=tab; p<tab+sizeof(tab)/sizeof(int); p++)
{
   std::cout << *p << " ";
}
std::cout << std::endl;
```

Het feit dat algoritmes zoals `sort()` zowel met pointers als met iterators van containers kan werken, ligt aan het feit dat deze algoritmes geschreven zijn als templatefuncties.

Als je een andere volgorde wil krijgen of als het gebruikte inhoudstype niet sorteerbaar is, moet je zelf een vergelijker bijvoegen. Hoe je dit doet, hebben we als gedemonstreerd bij de `set` container.

1. Gegeven is een lijst van rechthoeken. Vergelijk elke rechthoek met elke andere rechthoek en ga na of er een overlapping is. Tel het aantal overlappingen.
2. Gegegeven is een lijst van rechthoeken. Maak 3 nieuwe lijsten: één voor de rechthoeken die links van een verticale lijn staan, één voor de rechthoeken rechts van deze lijn en één voor rechthoeken die over de lijn staan. Overloop alle rechthoeken uit de gegeven lijst en plaats ze in één van de 3 nieuwe lijsten.
3. In een boek zijn er meerdere pagina's die elk bestaan uit meerdere woorden. Ontwerp hiervoor de klassen `Boek`, `Pagina` en `Woord`. Maak in de klassen ook de methode `bouwIndex()`: deze methode bouwt een index van alle woorden. Maak hiervoor de klassen `Index` en `IndexElement`. De eerste klasse stelt de index voor met daarin meerdere `IndexElement` objecten. In de `IndexElement` klasse wordt een pointerverwijzing naar het `Woord` en een lijst van `Woord` bijgehouden.
4. In een school (klasse `School`) hebben docenten (klasse `Docent`) meerdere lessen (klasse `Les`) op hun uurrooster. Elke les heeft een begin- en een einduur. Maak in de klasse `School` een methode die een lijst opbouwt van alle docenten die aanwezig kunnen zijn op een vergadering die start op een beginuur en loopt tot een einduur.

De C++11 en C++14 standaarden zijn een mijlpaal in de evolutie van C++. Eén van de belangrijkste aanvullingen in de C++11 standaard is de lambdafunctie. Met deze constructie kan je anonieme functies maken die als parameter aan een andere functie kunnen meegegeven worden. Dit is heel handig als je containers en algoritmes wil parametreren met een bepaald gedrag. In dit deel worden o.a. de lambdafuncties, en hoe je die toepast bij algoritmes, uitgelegd.

`auto` en `decltype`

Met het sleutelwoord `auto` kan je kortere declaraties van variabelen schrijven. Een variabele die met `auto` wordt gedeclareerd moet wel van ergens een type krijgen. Meestal komt dit van de initialisatiewaarde. De compiler analyseert deze waarde en leidt hieruit het benodigde type af. Indien de compiler hierin niet slaagt, komt er een foutmelding. Het gebruik van `auto` betekent helemaal niet dat het type mag ontbreken. C++ is een taal met stricte types: elke variabele moet een type krijgen.

Hier zijn de eerste voorbeelden:

```
auto x = 56;
auto y = 123L;
auto z = 123.5;
auto p = &x;
// auto q; // fout, initialisatie ontbreekt

std::cout << "type x " << typeid(x).name() << std::endl;
std::cout << "type y " << typeid(y).name() << std::endl;
std::cout << "type z " << typeid(z).name() << std::endl;
std::cout << "type p " << typeid(p).name() << std::endl;
```

Als je dit fragment uitvoert met het volgende commando:

```
./testauto | c++file -t
```

Dan zie je de types die de compiler afgeleid heeft.

En dit is de output:

```
type long long int
type unsigned long long long
type ... double
type p int*
```

Ook bij het terugkeertype van functies kan `auto` gebruikt worden.

```
auto som(int a, int b)
{
   return a + b;
}

auto s = som(1, 2);
```

Hier wordt het type bepaald door de parameters die in de returnuitdrukking voorkomen. Ja mag `auto` niet toepassing bij de parameters zelf.

Bij het schrijven van prototypes kan het handig zijn om het returntype achteraan te schrijven. Vooraan schrijf je dan `auto`. Hier geeft `auto` aan dat het type nog volgt.

```
auto som(int, int) -> int;
```

Bij lambdafuncties geeft `auto` een kortere schrijfwijze van het type.

```
auto f = []() { return 5; };
```

Het volgende fragment is een lambdafunctie die volledig generiek is; nergens wordt een concreet type vermeld. De enige voorwaarde is dat het uiteindelijke type een vermenigvuldiging moet kennen. Als het geen rekenkundig type is, moet er een `operator*()` bestaan.

```
auto h = [](auto x) { return 3*x; };
```

Met `decltype` kan je het type van een andere uitdrukking overnemen. In het volgende voorbeeld is het terugkeertype ook `int`.

```
auto dubbel(int g) -> decltype(g)
{
   return 2*g;
}
```

Het terugkeertype is hier hetzelfde type als dat van de parameter `g`.

Constante uitdrukkingen

In het verleden, voor de opgang van C++, werden constanten gemaakt met `#define`. Deze regel wordt verwerkt met de preprocessor. Alle regels die starten met een `#` worden hiermee verwerkt. Hier is een

voorbeeld:

```
#define AANTAL  10
```

Het nadeel van deze schrijfwijze is dat er geen typecontrole is. Met `#define` gebeurt er een tekstvervanging. Overal waar `AANTAL` voorkomt, wordt dit vervangen door `10`. Een betere notatie is de volgende:

```
const int aantal = 10;
```

In dit geval is er wel een type vastgelegd. Om zeker te zijn dat de constante al bekend is tijdens de compilatie, is er nu het sleutelwoord `constexpr`. Hiermee geef je aan dat de waarde zeker bij de compilatie moet bekend zijn. Het gebruik is dan zo:

```
constexpr int aantal = 10;
```

Het verschil tussen `const` en `constexpr` is subtiel en niet altijd waarneembaar. In veel notaties gedraagt `const` zich als `constexpr` omdat moderne compilers als optimalisatie tijdens de compilatie al zoveel mogelijk proberen de `const` uitdrukkingen uit te rekenen.

Het is mogelijk om functies `constexpr` te maken. Dan worden de functies tijdens de compilatie uitgevoerd. In C++11 hadden deze functies grote beperkingen; alleen een `return` mag erin voorkomen, `if`, `for` en `while` waren verboden. In C++14 zijn die beperkingen er niet meer. Hier is een voorbeeld van een C++11 `constexpr` functie die de som van de getallen van 1 tot n uitrekent.

```
constexpr int som(const int n)
{
   return n==0 ? 0 : n + som(n-1);
}
```

Er wordt gebruik gemaakt van de voorwaardelijke uitdrukking, een notatie die nog uit het C tijdperk stamt. Als `n` `0` is, wordt een `0` teruggegeven. Anders wordt het resultaat van een recursieve uitdrukking teruggegeven. je kan de functie zo gebruiken:

```
int tab[som(5)];
std::array<int, som(5)> tab2;
```

Hier geeft het gebruik van `constexpr` de zekerheid dat de constante al bij de compilatie bekend is en dat ze kan gebruikt worden bijvoorbeeld als dimensie van een array.

De `array` en `tuple` container

Beide klasse zijn in STL bijgevoegd sinds de C++11 standaard.

De `array` container

De `array` klasse is een container die intern met C array voor de opslag zorgt. Deze array heeft een vaste afmeting die achteraf niet meer gewijzigd kan worden. Deze eigenschap is het gevolg van het interne gebruik van de C array. Er wordt bij deze container geen gebruik gemaakt van dynamisch geheugen dat met `new` wordt gereserveerd. Dit maakt de `array` bijzonder geschikt voor het gebruik op platformen waar het beschikbare geheugen zo klein is dat er geen heap bestaat. C++ inzetten op platformen zonder heap (bijvoorbeeld low-end 32 bit embedded system) vraagt een speciale aanpak waar de `array` zeker zijn plaats heeft.

Een mogelijke implementatie zou er zo kunnen uitzien:

```
template<class T, int N>
class Array
{
private:
   T tab[N];

public:
   T &operator[](int index)
   {
```

```
      return tab[index];
   }
};
```

De opslagruimte `T tab[N]` is een gewone C array, `T` is het inhoudstype en `N` is de dimensie van de array. Deze laatste moet constant zijn tijdens de compilatie.

Het gebruik van de `array` container ziet er zo uit:

```
#include <array>

std::array<int, 6> tab;

for (int i=0; i<5; i++)
{
   tab[i] = i*i;
}
```

In dit fragment wordt de `operator()` gebruikt voor de toegang tot de elementen. Je kan ook gebruik maken van de `at()` methode.

```
try
{
   for (int i=0; i<50; i++)
   {
      std::cout << tab.at(i);
   }
}
catch (std::out_of_range)
{
   std::cout << "out of range" << std::endl;
}
```

Deze aanpak heeft als voordeel dat de index gecontroleerd wordt; bij de `operator()` gebeurt dat niet.

De initialisatie van een `array` kan met `{ }` geschreven worden, met of zonder isteken. Deze declaratieve schrijfwijze maakt de opbouw van de array compacter.

```
std::array<int, 5> tab = {1, 2, 3, 4, 5};
std::array<int, 5> tab2 {1, 2, 3, 4, 5};
```

Het volgende voorbeeld is iets langer en toont hoe de declaratieve initialisatie te hulp komt om een array als datalid te initialiseren. Een extra eis bij dit voorbeeld is dat het moet kunnen werken op een platform zonder heap. Meer en meer zien we C++ opduiken als programmmeertaal voor low-end 32 bit platformen die beperkte bronnen hebben. Omdat de hoeveelheid beschikbare RAM zeer klein is, is de heap dan ook zeer klein of onbestaand. Dit betekent dat `new` dan ook niet gebruikt kan worden. Het voorbeeld hoe je in die situatie met lijsten en objecten kan werken.

Dit voorbeeld is de uitwerking van de probleemstelling dat een `Lijn` altijd bestaat uit 4 `Punt` objecten. Bij de uitwerking kiezen we ervoor om geen `new` te gebruiken om objecten te maken. Hierdoor is de `vector` container al meteen uitgesloten; intern maakt deze gebruik van `new` om de opslagruimte te kunnen vergroten. De `array` is wel bruikbaar omdat `new` hierin ontbreekt.

```
#include <iostream>
#include <array>

class Punt
{
private:
   int x;
   int y;

public:
   Punt(int ix, int iy) : x(ix), y(iy)
```

```
   {
   }
};

class Lijn
{
private:
   std::array<Punt, 4> punten;

public:
   Lijn(Punt ip1, Punt ip2, Punt ip3, Punt ip4) :
      punten({ip1, ip2, ip3, ip4})
   {
   }
};


int main()
{
   Lijn lijn1(Punt(1,2), Punt(3,4), Punt(5,6), Punt(7,8));

   std::cout << "sizeof(lijn1) " << sizeof(lijn1) << std::endl;
   // geeft 32 = 2*4*4, dus geen heapruimte in gebruik
}
```

Hier volgt de beschrijving van het voorbeeld.

We maken 2 klassen `Punt` en `Lijn`. Een `Punt` bestaat uit een `x` en `y` en een `Lijn` bestaat uit 4 `Punt` objecten. De datamember voor de opslag van de 4 `Punt` objecten ziet er zo uit:

```
std::array<Punt, 4> punten;
```

De initialisatie van de `punten` array gebeurt in de constructor van `Lijn`.

```
Lijn(Punt ip1, Punt ip2, Punt ip3, Punt ip4) :
   punten({ip1, ip2, ip3, ip4})
{
}
```

Hier wordt `punten` geïnitialiseerd met `{ip1, ip2, ip3, ip4}`. Dat wordt dan zo geschreven:

```
punten({ip1, ip2, ip3, ip4})
```

Dit is dezelfde techniek die we al eerder gebruikt hebben bij de eerste `array` voorbeelden maar nu is dat toegepast op de initialisatie van dataleden van een klasse met behulp van een constructor.

In `main()` wordt de `lijn1` variabele aangemaakt. Bij de initialisatie worden 4 parameters in het formaat `Punt(1, 2)` doorgegeven. Dit zijn tijdelijke objecten waarvan de waarde wordt doorgegeven aan de constructor.

```
int main()
{
   Lijn lijn1(Punt(1,2), Punt(3,4), Punt(5,6), Punt(7,8));

   std::cout << "sizeof(lijn1) " << sizeof(lijn1) << std::endl;
   // geeft 32 = 2*4*4, dus geen heapruimte in gebruik
}
```

Tot slot staat er in `main()` nog een `sizeof(lijn1)`. Hiermee wordt de totale benodigde geheugenruimte in bytes van `lijn1` berekend. Dit geeft 32 als resultaat. Elk `Punt` heeft 2 `int`'s van elk 4 bytes. Dat maakt 8 bytes per `Punt`. Er zijn 4 `Punt`'en, dus 4*8 geeft 32 bytes. Door deze berekening zien we dat de volledige benodigde geheugenruimte in de klassen zit en dat er geen extra heapruimte wordt gebruikt. Hiermee is bewezen dat een `array` geen heap gebruikt.

De `tuple` container

Met een `tuple` kan je structuren van heterogene types maken zonder dat je hiervoor een `class` of `struct` moet maken. Een `tuple` moet je parametreren met de types van de gewenste datamembers. In het volgende voorbeeld hebben we een `tuple` bestaande uit 2 `int`'s en 1 `char>`.

```
#include <tuple>

std::tuple<int, int, char> tu1 = std::make_tuple(3, 6, 'A');
```

Je moet een `tuple` altijd maken met `std::make_tuple()`. Voor elk type uit de declaratie moet je een parameter meegeven. In dit voorbeeld geven we de waarden `3`, `6` en `'A'` mee.

Met de `get()` functie kan de de afzonderlijke waarden van de tuple opvragen. Deze functie is een templatefunctie en dit betekent dat de paramater met kleiner en groter dan tekens moet meegegeven worden. Dit betekent dan ook dat de waarde tijdens de compilatie moet bekend zijn.

```
std::cout << std::get<0>(tu1) << std::endl;
std::cout << std::get<1>(tu1) << std::endl;
std::cout << std::get<2>(tu1) << std::endl;
```

Een poging om het zo te schrijven gaat niet lukken.

```
int letter = 2;
std::cout << std::get<letter>(tu1) << std::endl; // Fout
```

Dit zal wel kunnen omdat we een `constexpr` maken.

```
constexpr int letter = 2;
std::cout << std::get<letter>(tu1) << std::endl; // ok
```

Het is mogelijk om met dezelfde `get()` functie de `tuple` te wijzigen.

```
std::get<0>(tu1) = 4;
```

De `get()` geeft immers een referentie naar de waarde terug. Dit betekent dat de waarde kan gewijzigd worden.

De volgende werkwijze laat toe om bestaande variabelen te combineren tot een `tuple`. Merk op dat de `tuple` nu referenties naar types bijhoudt.

```
int  n;
bool b;

std::tuple<int&, bool&> val = std::tie(n, b);
```

Dit betekent dan ook dat de ondersteunende variabelen `n` en `b` moeten blijven bestaan zolang ook `val` bestaat. De groepering doen we met de `tie()` functie.

Een `tuple` is vooral handig om meer dan één waarde vanuit een functie terug te geven. De volgende functie geeft een `int` en een `bool` terug.

```
std::tuple<int, bool> doe()
{
   return std::make_tuple(15, false);
}
```

Vanaf C++17 is het mogelijk om de terug te geven tuple zo te maken:

```
std::tuple<int, bool> doe2()
{
   return {15, true};
}
```

Hier wordt getoond hoe de teruggegeven tuple kan verwerken. Met `tie()` wordt een nieuwe tuple gemaakt die het resultaat van de functie opvangt.

```
std::tie(n, b) = doe();
std::cout << "{" << n << ", " << b << "}" << std::endl;
```

Vooral het gebruik als returntype maakt de `tuple` aantrekkelijk.

Smart pointers

Smart pointers bestaan al een tijdje in C++ en worden gebruikt om de kans op geheugenlekken te verkleinen. Wie dynamisch objecten maakt met `new` heeft ook de verantwoordelijkheid om het gereserveerde geheugen vrij te geven met `delete`. Wanneer dit niet gebeurt, is er sprake van een geheugenlek. In programma's die maar een korte tijd draaien zijn geheugenlekken niet echt een probleem maar in programma's die permanent draaien kunnen we ons geen geheugenlekken permitteren.

Als we de klassieke C pointer gebruiken, is de werkwijze als volgt:

```
Punt * p = new Punt(3, 4);

...

delete p;
```

Wanneer de reservatie en vrijgaven in hetzelfde bloc staan, is een visuele controle op geheugenlekken van de broncode eenvoudig. Als pointers als resultaten van functies teruggegeven worden en dan als parameter doorgegeven worden aan andere functies, is het moeilijk om de broncode te controleren op geheugenlekken. In vele gevallen circuleren er ook nog kopieën van pointers en dit maakt het lastig om geheugenlekken te vermijden.

Een mogelijke strategie is zoveel mogelijk pointers bijhouden als datamember in de klassen. In dat geval neemt de destructor de vrijgave van de poiners voor zijn rekening. De klasse heeft dan het eigendomsrecht van de pointer en is dus verantwoordelijk voor de vrijgave.

Een andere oplossing is het gebruik van de smart pointers `unique_ptr` en `shared_ptr`. C++ kent geen automatische garbage collection zoals Java maar met de smart pointers is er een oplossing die het comfort van de garbage collection goed benadert.

In C++ bestond al eerder de `auto_ptr` maar deze is met de komst van C++11 en C++14 verouderd en mag je niet meer gebruiken.

In bepaalde gevallen moet je gebruik maken van de `weak_ptr`. Dit is een smart pointer die de mogelijkheid heeft om de verwijzing naar het aangewezen object te verliezen. De smart pointers `unique_ptr` en `shared_ptr` hebben de eigenschap dat zolang de smart pointer bestaat, zal het aangewezen object ook blijven bestaan. Bij de `weak_ptr` is dat niet het geval; je moet bij elk gebruik van de `weak_ptr` nagaan of het aangewezen object nog bestaat. Wat alle smart pointers als gemeenschappelijk kenmerk hebben is de automatische vrijgave van het aangewezen object.

Verwijzingen met `unique_ptr`

Zoals de naam het zegt, heb je bij de `unique_ptr` nooit meer dan één pointer die een verwijzing naar het aangewezen object bijhoudt. De overige `unique_ptr`'s hebben `nullptr` als waarde. Het is wel mogelijk om de verwijzing van de ene `unique_ptr` naar de andere `unique_ptr` door te geven. Door dit mechanisme is de vrijgave van het aangewezen object eenvoudig: alleen de pointer die als laatste de verwijzing in handen gekregen heeft, doet de vrijgave.

Een `unique_ptr` maak je zo:

```
std::unique_ptr<Punt> p = std::make_unique<Punt>(7,8);
```

Dit is de officiële werkwijze; ze garandeert dat de vrijgave van het object correct zal verlopen, ook bij exceptions. De volgende schrijfwijze geeft die garantie niet en is daarom te vermijden:

```
std::unique_ptr<Punt> p2 = new Punt(7,8); // Fout, mag niet
std::unique_ptr<Punt> p3(new Punt(7,8)); // Niet fout, maar niet veilig
```

Het aanmaken van `p2` is verboden en het aanmaken van `p3` is niet verboden maar is onveilig. Als er een exception optreedt tijdens het uitvoeren van de `new` bestaat er gevaar op geheugenlekken. Met de officiële schrijfwijze bestaat dat gevaar niet.

Het volgende fragment is ook problematisch:

```
Punt p4(5, 6);
std::unique_ptr<Punt> p5(&p4);
```

Hier maken we eerst een lokaal object `p4` (een object dat zijn geheugenruimte op stack krijgt) en daarna nemen we het adres van `p4` en we houden dat bij in `p5`. Beide regels zullen wel correct uitgevoerd worden maar wanneer de levensduur van `p5` eindigt, wordt automatisch het aangewezen object, in dit geval `p4` vrijgegeven. Dit kan niet omdat `p4` nooit met `new` is gereserveerd.

Een andere beperking is dat het kopiëren van de `unique_ptr` smart pointer niet zomaar gaat. Je kan eigenlijk niet kopiëren maar enkel verplaatsen. Dit laatste betekent dat je de verwijzing naar het object kan verplaatsten van de ene `unique_ptr` naar de andere. De bronpointer wordt nul en de doelpointer neemt de verwijzing over. Op deze manier is er altijd maar één smart pointer die een verwijzing naar het object heeft. En dit zorgt ervoor dat de vrijgave van het object gemakkelijk kan georganiseerd worden.

Zomaar kopiëren mag niet.

```
auto p6 = p; // Fout
```

Verplaatsen mag wel.

```
std::unique_ptr<Punt> p7(std::move(p));
std::unique_ptr<Punt> p8 = std::move(p);
```

Beide schrijfwijzen hierboven zijn correct. In beide gevallen gebeurt er een verplaatsing. De `std::move()` notatie zorgt hiervoor. Het gebruik van deze notatie wijst op de zogenaamde *move*-semantiek. Dit is een werkwijze waarbij informatie niet gekopiëerd wordt maar wel verplaatst. Later zal dit verder uitgelegd worden.

De verplaatsingen van `p` naar `p7` en `p8` vragen toch wat extra uitleg. Dit zijn de stappen:

- De pointer `p` heeft een verwijzing naar een `Punt` object.
- Bij het aanmaken van `p7` wordt deze pointer geïnitialiseerd met `p`. Dit betekent dat de verwijzing doorgegeven wordt van `p` naar `p7`. Daarna verliest `p` de verwijzing; `p` krijgt de waarde `nullptr`.
- In een volgende stap wordt `p8` aangemaakt. Deze krijgt de waarde van `p` die ondertussen `nullptr` is.
- Uiteindelijk is `p7` de enige pointer met een verwijzing die niet `nullptr` is. De pointers `p` en `p8` hebben wel `nullptr` als waarde. Dat kan je zo testen:

  ```
  p7->toon();
  if (p8 != nullptr)
  {
     p8->toon();
  }
  else
  {
     std::cout << "p8 is null\n";
  }
  ```

In het verdere gebruik is er geen verschil tussen de klassieke C pointer en de `unique_ptr`. Je kan de notatie `p7->toon()` om een methode te starten blijven gebruiken. Alleen de `delete` aan het einde van levensduur van de `unique_ptr` zal nu automatisch uitgevoerd worden.

Als besluit kan gesteld worden dat het gebruik van de `unique_ptr` aan te bevelen is in situaties waar meerdere kopieën niet noodzakelijk zijn. Het grote voordeel is ontbreken van geheugenlekken. Wanneer we toch meerdere gelijktijdige kopieën willen beheren, moeten we overschakelen naar de `shared_ptr`.

Verwijzingen met `shared_ptr`

Met `shared_ptr` verwijzingen kan je meerdere verwijzingen naar hetzelfde object bijhouden zonder dat je zelf de vrijgave moet doen. Intern wordt er een verwijzingenteller gebruikt die bijhoudt hoeveel verwijzingen er nog actief zijn. Het voordeel is dat je geen geheugenlekken krijgt maar het nadeel is dat er meer geheugenverbruik nodig is om de boekhouding van het aantal verwijzingen bij te houden. Hou ook

rekening met het feit dat `shared_ptr` verwijzingen kopiëren trager is dan `unique_ptr` kopiëren. Bij elke kopie wordt de boekhouding van het aantal verwijzingen aangepast en dat kost tijd.

Hier is een compleet voorbeeld met `shared_ptr` verwijzingen.

```
#include <iostream>
#include <string>
#include <memory>

class Punt
{
private:
   int x;
   int y;

public:
   Punt(int ix, int iy) : x(ix), y(iy)
   {
      std::cout << "Punt()\n";
   }
   ~Punt()
   {
      std::cout << "~Punt(" << x << "," << y <<")\n";
   }
   void toon()
   {
      std::cout << "Punt " << x << "," << y <<"\n";
   }
};


void toonpunt(std::string naam, const std::shared_ptr<Punt> &p)
{
   if (p != nullptr)
   {
      std::cout << naam << " "; p->toon();
   }
   else
   {
      std::cout << naam << " nullptr" << std::endl;
   }
}


int main()
{
   std::cout << "test pointers\n";

   // Dit mag niet
   //std::shared_ptr<Punt> p = new Punt(3,4);

   // Dit mag wel
   std::shared_ptr<Punt> p2(new Punt(3,4));

   // Dit mag ook
   std::shared_ptr<Punt> p3 = std::make_shared<Punt>(7,8);

   // kopieren mag wel
   auto p4 = p3;
```

```
    // move mag wel
    std::shared_ptr<Punt> p5(std::move(p3));
    std::shared_ptr<Punt> p6 = std::move(p3);

    toonpunt("p2", p2);
    toonpunt("p3", p3); // == nullptr
    toonpunt("p4", p4);
    toonpunt("p5", p5);
    toonpunt("p6", p6); // == nullptr

    // geeft 2
    std::cout << "p4.use_count() " << p4.use_count() << std::endl;

    std::shared_ptr<Punt> p7 = std::make_shared<Punt>(9,10);
    p7 = std::make_shared<Punt>(11,12);
    toonpunt("p7", p7);

    std::shared_ptr<Punt> p8 = std::make_shared<Punt>(13,14);
    toonpunt("p8", p8);
    p8.reset();
    toonpunt("p8", p8);
}
```

En hier volgt de uitleg. De volgende manier om een object te maken mag je niet gebruiken. De compiler geeft een foutmelding.

```
std::shared_ptr<Punt> p = new Punt(3,4);
```

De volgende manier mag wel maar wordt afgeraden omdat er toch een kleine kans op een geheugenlek bestaat.

```
std::shared_ptr<Punt> p2(new Punt(3,4));
```

Deze manier is de juiste:

```
std::shared_ptr<Punt> p3 = std::make_shared<Punt>(7,8);
```

Kopiëren is geen probleem:

```
auto p4 = p3;
```

Verplaatsen mag ook. Hierbij wordt de bronverwijzing nul.

```
std::shared_ptr<Punt> p5(std::move(p3));
std::shared_ptr<Punt> p6 = std::move(p3);
```

Je kan opvragen hoeveel maal een object nog aangewezen wordt.

```
std::cout << "p4.use_count() " << p4.use_count() << std::endl;
```

In het volgende fragment wijst `p7` eerst naar 9,10 en daarna naar 11,12. Het `Punt` met 9,10 wordt dan eerst automatisch vrijgegeven.

```
std::shared_ptr<Punt> p7 = std::make_shared<Punt>(9,10);
p7 = std::make_shared<Punt>(11,12);
```

Met de methode `reset()` wordt een verwijzing nul gemaakt en het aangewezen object wordt mogelijk vrijgegeven.

```
std::shared_ptr<Punt> p8 = std::make_shared<Punt>(13,14);
p8.reset();
```

Het enige probleem dat bij het gebruik van `shared_ptr` verwijzingen kan optreden, zijn de circulaire verwijzingen: een object a wijst naar het object b en b op zijn beurt verwijst naar a. In dat geval heeft elk object één verwijzing en zal er geen vrijgave van de objecten gebeuren op het moment dat de verwijzingen niet meer bestaan.

Het volgende programma demonstreert dit probleem. Er is sprake van de klassen `A` en `B`. Van elk wordt één object gemaakt dat telkens naar het andere object wijst. Om het voorbeeld eenvoudig te houden zijn alle datamembers `public`.

```
#include <iostream>
#include <string>
#include <memory>

class B;

class A
{
public:
   std::shared_ptr<B> b;

   A()
   {
      std::cout << "A()" << std::endl;
   }
   ~A()
   {
      std::cout << "~A()" << std::endl;
   }
};


class B
{
public:
   std::shared_ptr<A> a;

   B()
   {
      std::cout << "B()" << std::endl;
   }
   ~B()
   {
      std::cout << "~B()" << std::endl;
   }
};



int main()
{
   std::cout << "test pointers\n";

   std::shared_ptr<A> a1 = std::make_shared<A>();
   std::shared_ptr<B> b1 = std::make_shared<B>();

   a1->b = b1;
   b1->a = a1;

   // geen van beide pointers a1 en b1 worden vrijgegeven
}
```

Bij het verlaten van de `main()` functie bestaan de verwijzingen `a1` en `b1` niet meer en zouden we verwachten dat beide aangewezen objecten vrijgegeven zullen worden maar dat is niet het geval.

Als je dit probleem wil oplossen, moet je gebruik maken van `weak_ptr`. Dit is een type verwijzing dat

onverwacht de verwijzing naar het object kan verliezen.

Verwijzingen met `weak_ptr`

De `weak_ptr` verwijzingen hebben de eigenschap dat het aangewezen object plotseling vrijgegeven wordt. De `weak_ptr` verwijzingen tellen dus niet mee bij de boekhouding van het aantal verwijzingen naar een object. Dit type verwijzing kan je gebruiken als je bijvoorbeeld een lijst van alle objecten van een bepaalde klasse wil bijhouden zonder dat dit invloed heeft op de levensduur van de objecten. Als de lijst zou bestaan uit `shared_ptr` verwijzingen worden de objecten nooit vrijgegeven zolang de lijst bestaat. En dat willen we niet in deze toepassing.

Hier is het gebruik van de `weak_ptr` verwijzing.

```
std::shared_ptr<Punt> p9 = std::make_shared<Punt>(15,16);
std::weak_ptr<Punt> p10(p9);
if (p10.expired())
{
   std::cout << "p10 expired" << std::endl;
}
else
{
   std::cout << "p10 niet expired" << std::endl;
}

p9 = nullptr;
std::shared_ptr<Punt> p11 = p10.lock();
if (p11 == nullptr)
{
   std::cout << "p11 nullptr" << std::endl;
}
else
{
   std::cout << "p11 niet nullptr" << std::endl;
}
```

In het eerste deel van het fragment worden `p9` en `p10` gemaakt. `p10` wordt met `p9` geïnitialiseerd. `p10` is een `weak_ptr` verwijzing die de wijzer naar het object kan verliezen. Dat zou alleen maar het geval zijn als de variabele `p9` niet meer zou bestaan. Maar hier is dat niet zo. `p9` bestaat nog en heeft voorlopig een geldige verwijzing. Hierdoor zal de test `p10.expired()` een `false` als resultaat geven, dus `p10` heeft ook een geldige verwijzing.

Halverwege het fragment wordt `p9` nul gemaakt met `p9 = nullptr;`. De volgende stap is het opvragen van een `shared_ptr` met `p10.lock()`. Als op dat moment `p10` geen geldige verwijzing heeft (wat hier inderdaad het geval is), zal `p11` de waarde `nullptr` hebben. De `lock()` methode laat toe om een kopie te verkrijgen van een `weak_ptr`. Als de `weak_ptr` nul is, zal de kopie, die een `shared_ptr` is, ook nul zijn. Als de `weak_ptr` op het moment van de kopie niet nul is, zal de `shared_ptr` kopie ook niet nul zijn en er is de zekerheid dat de `shared_ptr` kopie zijn verwijzing niet kan verliezen.

Hier is het voorgaande programma met de circulaire verwijzingen hernomen. Het enige verschil is dat de verwijzing die in de klasse `B` wordt bijgehouden wordt, nu van het type `weak_ptr` is.

```
#include <iostream>
#include <string>
#include <memory>

class B;

class A
{
public:
   std::shared_ptr<B> b;
```

```
   A()
   {
      std::cout << "A()" << std::endl;
   }
   ~A()
   {
      std::cout << "~A()" << std::endl;
   }
};


class B
{
public:
   std::weak_ptr<A> a;

   B()
   {
      std::cout << "B()" << std::endl;
   }
   ~B()
   {
      std::cout << "~B()" << std::endl;
   }
};



int main()
{
   std::cout << "test pointers\n";

   std::shared_ptr<A> a1 = std::make_shared<A>();
   std::shared_ptr<B> b1 = std::make_shared<B>();

   a1->b = b1;
   b1->a = a1;

   // beide pointers a1 en b1 worden vrijgegeven
}
```

Nu worden beide objecten correct vrijgegeven. Als er circulaire verwijzingen binnen de klassen ontstaan, moet je de cirkel doorbreken door `weak_ptr`'s te gebruiken. Het is dan handig om het klassediagramma te tekenen en de verwijzingen tussen de klassen als pijlen voor te stellen. Op deze manier kan je snel cirkels ontdekken.

Rvalue referenties

Sinds C++11 is er naast het bestaande referentietype een nieuw referentietype bijgekomen. Het oude referentietype wordt nu *lvalue referentietype* genoemd en het nieuwe type heet *rvalue referentietype*. Dit nieuwe type maakt het mogelijk om in specifieke gevallen geen geheugen te verplaatsen in in de plaats daarvan alleen maar de verwijzing naar het geheugen te kopiëren. Deze werkwijze heet *move semantiek*.

Sinds het ontstaan van C en ook al daarvoor in het tijdperk waar assembler gebruikt werd, had de programmeur altijd inzicht op het geheugenmodel van de ontwikkelde programma's. De toenmalige programmeurs hadden bij de omschakeling naar C bijna altijd een voorgeschiedenis in assembler. En bij deze laatste taal kan je niet anders dan rekening houden met de wijze waarop een programma het geheugen gebruikt. Een C programmeur weet dus altijd dat:

- elke variabele heeft geheugen nodig
- er zijn parameters, globale, lokale variabelen
- parameters en lokale variabelen krijgen geheugen van de stack
- globale variabelen krijgen geheugen uit het datasegment
- je kan het adres nemen van een variabele
- een adres van een variabele kan je opslaan in een pointer

Uiteraard zijn dit bekende concepten. En tijdens het programmeren hoef je daar niet zwaar over na te denken. Pointers worden gebruikt om adressen op te slaan, zoals in het volgende fragment.

```
int a;
int *p = &a;
```

Het referentietype in C++ is een andere en ook abstractere manier om met het geheugen om te gaan. Hier is een fragment:

```
int b;
int &c = b;
```

De variabele `c` heeft geen eigen geheugenallocatie maar gebruikt de geheugenruimte van `b`. Intern wordt dit met een pointer opgelost, alleen zie je dat niet kan kan je bijgevolg het opgeslagen adres in `c` niet manipuleren. Hierdoor kunnen er geen pointerfouten ontstaan.

De reden waarom in C++11 het rvalue referentietype is bijgevoegd is enkel om efficiëntieredenen. Met dit type is het mogelijk om zelf bepaalde optimalisaties te verkrijgen. Voor de doorsnee programma's is dit misschien niet zo belangrijk maar voor de ontwerpers van STL is dit wel echt belangrijk. Door gebruik te maken van het rvalue referentietype is de STL wel sneller geworden. Om dit type moeten we het verschil tussen rvalue en lvalue begrijpen.

Een vereenvoudigde definitie zegt dat een *lvalue* een uitdrukking is die zowel links als rechts van een toekennings-`=` mag voorkomen. Een *rvalue* mag enkel rechts voorkomen.

```
int d = 4;     // d is lvalue
int e = 5;     // e is lvalue
int f = d * e; // f is lvalue, d * e is rvalue
d * e = 17;    // fout, rvalue staat links van =
```

Alleen de laatste regel van het bovenstaande fragment is fout omdat een rvalue niet links van de `=` mag staan. Je merkt ook dat je van een rvalue geen adres kan nemen. In bepaalde gevallen is het herkennen van een lvalue niet zo evident. Dat is zo in het volgende fragment:

```
int teller = 0;

int &doe()
{
   return teller;
}

void main()
{
   doe()++;
}
```

Omdat de functie `doe()` als terugkeertype `int &` heeft, het lvalue referentietype, is het mogelijk om de teruggegeven variabele te wijzigen met een increment. Merk op dat de variabele `teller` in dit voorbeeld een globale variabele is. Dit geeft de garantie dat de variabele nog bestaat, ook al is de `doe()` afgelopen. Je mag hier geen lokale variabele teruggeven. Dit zou betekenen dat je dan een variabele kan wijzigen die niet meer bestaat. De volgende versie van `doe()` is dus dik fout:

```
// FOUT
int &doe()
{
```

```
   int lokaal;
   return lokaal;
}
```

Als er geen & zou staan, zou er geen probleem zijn maar nu is het terugkeertype het lvalue referentietype. In dit geval wordt achter de schermen het adres van een verdwenen variabele teruggegeven. Als je het foute fragment met pointers herschrijft, wordt de fout wellicht duidelijk.

```
// FOUT
int *doe()
{
   int lokaal;
   return &lokaal;
}
```

Om de werking van de rvalue referentie te begrijpen, bekijken we het volgende complete maar toch niet te grote voorbeeld. De klasse A maakt gebruik van het nieuwe rvalue referentietype. Er wordt gebruik gemaakt van de C++ eigenschap om een methode met een bepaalde naam meerdere malen te implementeren, telkens met andere parameters.

```
#include <iostream>

/*
   Dit is het voorbeeld om de move semantiek te verklaren.
   Er is een onderscheid tussen
      lvalue reference   int &
      rvalue reference   int &&
 */


class A
{
public:
   A() : waarde(new int[1])
   {
      std::cout << "A::A() " << waarde << std::endl;
   }
   A(int w) : waarde(new int[1])
   {
      std::cout << "A::A(int) " << waarde << std::endl;
      waarde[0] = w;
   }
   A(const A &a) : waarde(new int[1])
   {
      std::cout << "A::A(const A&) " << waarde << std::endl;
      waarde[0] = a.waarde[0];
   }
   A(A &&a) : waarde(a.waarde)
   {
      std::cout << "A::A(A&&)" << std::endl;
      a.waarde = nullptr;
   }
   A &operator=(const A &a)
   {
      std::cout << "A::operator=(const A&)" << std::endl;
      waarde[0] = a.waarde[0];
      return *this;
   }
   A &operator=(A &&a)
   {
```

```
      std::cout << "A::operator=(A&&)" << std::endl;
      std::cout << "delete " << waarde << std::endl;
      delete [] waarde;
      waarde = a.waarde;
      a.waarde = nullptr;
      return *this;
   }
   A operator+(const A &a)
   {
      std::cout << "A::operator+(const A&)" << std::endl;
      std::cout << "maak sum\n";
      A sum;
      sum.waarde[0] = waarde[0] + a.waarde[0];
      return sum;
   }
   ~A()
   {
      std::cout << "A::~A() " << waarde << std::endl;
      if (waarde != nullptr)
      {
         delete [] waarde;
      }
   }
private:
   int *waarde;
};

int main(int argc, char *argv[])
{
   std::cout << "maak a1\n";
   A a1;
// A::A() 0xb65c30

   std::cout << "maak a2\n";
   A a2 = a1;
// A::A(const A&) 0xb65c50

   std::cout << "maak a3\n";
   A a3;
// A::A() 0xb65c70

   std::cout << "kentoe a3 = a1\n";
   a3 = a1;
// A::operator=(const A&)


   std::cout << "maak a4\n";
   A a4 = 7;
// A::A(int) 0xb65c90

   std::cout << "maak a5\n";
   A a5 = 6;
// A::A(int) 0xb65cb0

   std::cout << "maak a6\n";
   A a6;
// A::A() 0xb65cd0
```

```
   std::cout << "telop a6\n";
   a6 = a4 + a5;
// A::operator+(const A&)
// maak sum
// A::A() 0xb65cf0
// A::operator=(A&&)
// delete 0xb65cd0
// A::~A() 0

   std::cout << "maak a7\n";
   A a7;
// A::A() 0xb65cd0

   std::cout << "kentoe a7\n";
   a7 = std::move(a6);
// A::operator=(A&&)
// delete 0xb65cd0

   std::cout << "einde\n";
// A::~A() 0xb65cf0
// A::~A() 0
// A::~A() 0xb65cb0
// A::~A() 0xb65c90
// A::~A() 0xb65c70
// A::~A() 0xb65c50
// A::~A() 0xb65c30
   return 0;
}
```

De klasse `A` is zodanig gemaakt dat ervan objecten kunnen gemaakt worden zonder `new` te gebruiken. Intern wordt een pointer naar een geheel getal bijgehouden.

```
class A
{

private:
   int *waarde;
};
```

De `waarde` pointer wijst naar een `int` die met `new` is gereserveerd. Dit betekent dat we bij het kopiëren van `A` objecten moeten vermijden om zomaar pointers te kopiëren. Het gevaar bestaat dat er geheugenlekken of dubbele vrijgaven ontstaat. De enige oplossing om deze problemen te vermijden is het vastleggen van de benodigde `operator()` methoden. In dit voorbeeld zijn dat `operator=()` en `operator+()`.

We overlopen alle methoden van de klasse `A`. Eerst hebben we de verschillende constructors. Elke variant van de constructor reserveert altijd geheugenruimte voor één `int`. Afhankelijk van de doorgegeven parameter is er ook nog een initialisatie met deze parameter.

```
A() : waarde(new int[1])
{
   std::cout << "A::A() " << waarde << std::endl;
}
```

Deze constructor krijgt geen parameter en wordt gebruikt als we een object zonder initialisatie maken. Er wordt een `int` gereserveerd en de waarde ervan blijft onbepaald.

```
A(int w) : waarde(new int[1])
{
   std::cout << "A::A(int) " << waarde << std::endl;
   waarde[0] = w;
}
```

Deze constructor verwerkt de initialisatie met een geheel getal.

```
A(const A &a) : waarde(new int[1])
{
   std::cout << "A::A(const A&) " << waarde << std::endl;
   waarde[0] = a.waarde[0];
}
```

In dit geval gebeurt de initialisatie door een kopie te maken van de `int` van een ander `A` object. Merk op dat we van dit laatste object een lvalue referentie hebben. Dit betekent dat dit object bereikbaar via een adres. Vermits we dit object niet willen wijzigen, is de paramater `const`.

```
A(A &&a) : waarde(a.waarde)
{
   std::cout << "A::A(A&&)" << std::endl;
   a.waarde = nullptr;
}
```

Deze constructor heeft net zoals de vorige een `A` object als parameter. In dit geval is het een object dat niet via een adres bereikbaar. Dit betekent dat het object in de oproep van de constructor niet verder kan gebruikt worden. Hierdoor hebben we de vrijheid om de pointer naar de `int` over te nemen. Om geen dubbele vrijgave te veroorzaken geven we de `waarde` pointer van de parameter de nulwaarde. Merk op dat er bij deze initialisatie geen kopie van de `int` is gemaakt. Enkel de pointer is gekopiëerd. Hier is sprake van de *move semantiek*.

Nu volgen de `operator()` methoden.

```
A &operator=(const A &a)
{
   std::cout << "A::operator=(const A&)" << std::endl;
   waarde[0] = a.waarde[0];
   return *this;
}
```

Deze operator neemt enkel de `int` waarde over door een kopie. Het object waarvan we kopiëren is een lvalue en blijft verder bestaan. Daarom wordt aan dit object niets gewijzigd.

```
A &operator=(A &&a)
{
   std::cout << "A::operator=(A&&)" << std::endl;
   std::cout << "delete " << waarde << std::endl;
   delete [] waarde;
   waarde = a.waarde;
   a.waarde = nullptr;
   return *this;
}
```

Deze operator wordt gestart wanneer we kopiëren van een rvalue object. Vermits de levensduur van dit object op het einde loopt, mogen we de pointer naar de `int` overgenomen worden. Eerst moet wel de huidige `int` met `delete` vrijgegeven worden. Opnieuw is er sprake van de move semantiek.

```
A operator+(const A &a)
{
   std::cout << "A::operator+(const A&)" << std::endl;
   std::cout << "maak sum\n";
   A sum;
   sum.waarde[0] = waarde[0] + a.waarde[0];
   return std::move(sum);
}
```

Deze operator is er enkel voor lvalue referentie als parameter. Er wordt een nieuw object `sum` gemaakt dat de som van de `int`'s krijgt. Dit `sum` object wordt teruggegeven. Hierbij wordt expliciet aangegeven dat de move semantiek moet toegepast worden zodat de `int` niet gekopiëerd wordt.

```
~A()
{
   std::cout << "A::~A() " << waarde << std::endl;
   if (waarde != nullptr)
   {
      delete [] waarde;
   }
}
```

Tijdens de levensduur van een `A` object kan het toch gebeuren dat `waarde` nul wordt. Daarom is de `if` noodzakelijk.

Hier start `main()`. Er wordt telkens een fragment getoond met de nodige uitleg. De regels in commentaar zijn de output van verschillende methoden. Op deze manier kunnen we precies volgen wat er gebeurt.

```
   std::cout << "maak a1\n";
   A a1;
// A::A() 0xb65c30
```

Het object `a1` wordt gemaakt.

```
   std::cout << "maak a2\n";
   A a2 = a1;
// A::A(const A&) 0xb65c50
```

Het object `a1` wordt gemaakt en geïnitialiseerd met `a2`. Dit laatste is een lvalue.

```
   std::cout << "maak a3\n";
   A a3;
// A::A() 0xb65c70
```

Het `a3` object wordt gemaakt.

```
   std::cout << "kentoe a3 = a1\n";
   a3 = a1;
// A::operator=(const A&)
```

Het object `a3` krijgt de waarde van `a1`. En `a1` is een lvalue.

```
   std::cout << "maak a4\n";
   A a4 = 7;
// A::A(int) 0xb65c90
```

Het `a4` object wordt gemaakt en krijgt de waarde `7`.

```
   std::cout << "maak a5\n";
   A a5 = 6;
// A::A(int) 0xb65cb0
```

Het `a5` object wordt gemaakt en krijgt de waarde `6`.

```
   std::cout << "maak a6\n";
   A a6;
// A::A() 0xb65cd0
```

Het `a6` object wordt gemaakt zonder bepaalde waarde.

```
   std::cout << "telop a6\n";
   a6 = a4 + a5;
// A::operator+(const A&)
// maak sum
// A::A() 0xd6b0f0
// som is 13
// A::A(A&&)
// A::~A() 0
// A::operator=(A&&)
// delete 0xd6b0d0
```

```
// A::~A() 0
```

a4 en a5 worden opgeteld en de som gaat naar a6. Binnen deze operator wordt de lokale variabele sum gemaakt. Dit object houdt de som bij. Van dit object wordt de pointer gekopieerd. Dus hier is sprake van de move semantiek. Het is wel nodig dit expliciet aan te geven. Dit doen we met std::move().

```
   std::cout << "maak a7\n";
   A a7;
// A::A() 0xb65cd0
```

Het object a7 wordt gemaakt.

```
   std::cout << "kentoe a7\n";
   a7 = std::move(a6);
// A::operator=(A&&)
// delete 0xb65cd0
```

a7 krijgt de waarde van a6. Dit is geen kopie maar wel een move. Vergelijk het resultaat met de eerdere toekenning a3 = a1. Daar was het wel een kopie.

```
   std::cout << "einde\n";
// A::~A() 0xb65cf0
// A::~A() 0
// A::~A() 0xb65cb0
// A::~A() 0xb65c90
// A::~A() 0xb65c70
// A::~A() 0xb65c50
// A::~A() 0xb65c30
```

En dit is het einde.

Als besluit kunnen we stellen dat door het rvalue referentietype de constructor en operator=() kunnen weten of ze te maken hebben met een lvalue of een rvalue parameter. Vermits de rvalue parameter op dat moment een een aflopende levenscyclus heeft, kan er een move in plaats van een kopie uitgevoerd worden. Hierdoor is het programma sneller. Hierbij wordt de mogelijkheid gebruikt om een methode meerdere malen te implementeren.

```
doe(A &a)
{
  // kopieer
}

doe(A &&a)
{
  // move
}
```

Het rvalue referentietype is vooral nuttig om de interne geheugenboekhouding van een klasse te optimaliseren. Bij eenvoudige programma's is dit type misschien niet noodzakelijk maar de kennis dat dit type bestaat, is belangrijk om te beseffen dat C++11 in staat is om de efficiëntie van programma's te verhogen.

Lambdafuncties

De basis van lambdafuncties

De lambdafuncties zijn een nieuwe toevoeging in C++11. Het grote verschil met pointers naar functies is dat lambdafuncties in staat zijn om de toestand van omringende variabelen te bevriezen. Bij pointers naar functies is dat niet mogelijk, bij functieobjecten gaat dat wel maar dan moet je telkens een nieuwe klasse uitschrijven. Met de lambdafuncties is er nu een compacte schrijfwijze ter vervanging van de oude constructies.

We geven hier enkele korte voorbeelden die de kracht van deze nieuwe C++ constructie zullen tonen.

```
vector<int> tab = { 3, 5, 7, 8, 9};
```

```
for_each(tab.begin(), tab.end(), [](const int &a) { std::cout << a << std::endl; });
```

In het bovenstaande voorbeeld wordt gebruik gemaakt van het `for_each` algoritme. Dit gaat een bepaalde actie uitvoeren op elk element van de doorgegeven lijst. De actie wordt vastgelegd door een lambdafunctie. Die ziet er zo uit:

```
[](const int &a) { std::cout << a << std::endl; }
```

Dit betekent dat er een anonieme functie gemaakt wordt met een referentie naar een constante `int` als parameter. De body van de functie maakt gebruik van de parameter: de waarde wordt op het scherm getoond. Op de rechte haken moet je voorlopig niet letten, die worden dadelijk uitgelegd.

Je kan een lambdafunctie opslaan en gebruiken.

```
auto fu = [](const int &a) { std::cout << a << std::endl; }
fu(5);
```

De variabele `fu` hierboven heeft `auto` als type. Hiermee bespaar je een complexe typeuitdrukking.

Je kan ook expliciet het returntype vermelden.

```
auto l1 =  [](const int &a)->bool { return a > 4; };
std::cout << l1(8) << std::endl;
std::cout << l1(2) << std::endl;
```

Hier is dat het `bool` type. Het returntype wordt na de pijl vermeld.

Nu komen de rechte haken. Die worden gebruikt om de context van één of meerdere variabelen te bewaren tot het moment waarop de lambdafunctie gestart wordt.

```
int grens = 100;
auto l2 =  [grens](const int &a)->bool { return a > grens; };
grens++;
std::cout << l2(80) << std::endl;
std::cout << l2(120) << std::endl;
```

Op het moment dat de lambdafunctie `l2` vastgelegd wordt, heeft de variabele `grens` de waarde `100`. Om aan te geven dat deze waarde ongewijzigd moet kunnen gebruikt worden binnen de body van de lambdafunctie, moet de variabelenaam binnen de rechte haken geplaatst worden. Na het vastleggen van de lambdafunctie mag je de waarde van `grens` wijzigen. Dit heeft geen effect op de waarde die binnen de body van de lambdafunctie gebruikt zal worden; daar blijft het steeds `100`. Bij de functionele programmeertalen noemt men dit principe *capture*.

Als je meerdere variabele wilt bewaren, moet je ze opsommen met een komma ertussen.

```
int  c1 = 45;
int  c2 = 67;
int  c3 = 89;
auto l3 =  [c1, c2, c3]() { return c1 + c2 + c3; };
```

Het gebruik van de rechte haken `[]` voor het vastleggen van de gebruikte variabelen uit de context noemt met *capture*. Deze capture bestaat in de meeste functionele programmeertalen en is er nu ook in C++.

Hier is nog een voorbeeld met een capture. In dit geval staat er een ampersand voor de naam van de variabele. Dit betekent dat de betrokken variabele `totaal` kan gewijzigd worden vanuit de body van de lambdafunctie.

```
int totaal = 0;
for_each(tab.begin(), tab.end(), [&totaal](const int &a) { totaal += a; });
std::cout << totaal << std::endl;
```

Het bovenstaande voorbeeld telt elk element van de lijst `tab` op bij `totaal`. Omdat `totaal` gewijzigd moet kunnen worden, moet je de capture schrijven met een ampersand: `[&totaal]`.

Er zijn verschillende manieren om de capture vast te leggen. Hier is een overzicht:

`[]` Er gebeurt geen capture.

[&] In de capture zitten alle lokale variabelen. Elke variabele is wijzigbaar.

[=] In de capture zitten alle lokale variabelen. Van elke variabele wordt een kopie gemaakt. [33] De originele variabelen zijn dus niet wijzigbaar.

[&totaal] De variabele `totaal` zit in de capture en kan gewijzigd worden.

[aantal] De variabele `aantal` zit als een kopie in de capture.

[this] De `this` pointer van de omringende klasse zit in de capture. Zo heb je toegang tot de datamembers van dit object[34].

Hier is een voorbeeld dat de capture van lokale variabele demonstreert.

```
void fu()
{
   int a = 1;
   int b = 2;
   int c = 3;

   auto verhoog = [&]() { a++; b++; c++;};
   auto toon = [=]() { std::cout << a << b << c << std::endl;};

   verhoog();
   toon();
}
```

De `verhoog` lambdafunctie doet een capture met de ampersand. Dit betekent dat alle lokale variabelen kunnen gewijzigd worden. Bij `toon` is er geen wijziging nodig, daarom wordt bij de capture een `=` teken vermeld.

En hier is een voorbeeld dat `this` in de capture gebruikt.

```
class Punt
{
public:
   Punt() : x(1), y(2) {}
   void toon()
   {
      // toon x en y
      [this](){ std::cout << x << " " << y << std::endl; }();
   }

private:
   int x;
   int y;
};

int main()
{
   Punt p;
   p.toon();
}
```

Het bovenstaande voorbeeld gebruikt de `this` capture om toegang te kunnen krijgen tot de datamembers `x` en `y`.

Het type van een lambdafunctie kan samengesteld worden met behulp van de `function()` functie. Hier is een voorbeeld:

[33] Denk eraan dat voor elke lambdafunctie met een capture de compiler een klasse genereert waarmee een functieobject wordt gemaakt. Met kopie wordt bedoeld dat de waarde van de lokale variabele gekopiëerd wordt naar een datalid van het functieobject.

[34] En kan je die datamembers dan ook wijzigen? Even nadenken!

```
function<void ()> verhoog = [&]() { a++; b++; c++;};
```

Je hoeft deze notatie niet te gebruiken; met `auto` gaat het ook.

Nu al kan gesteld worden dat de komst van lambdafuncties in een aantal gevallen het gebruik van STL zal vereenvoudigen. We zien dus dat C++ in dit geval een sterke invloed heeft ondergaan van de functionele programmeertalen. Als je lambdafuncties en andere moderne notaties wil gebruiken bij de ontwikkeling, moet je wel beschikken over een compiler die de C++11 standaard ondersteunt. Bij de GNU C++ compiler moet je de `--std=c++11` optie meegeven.

```
g++ --std=c++11
```

Lambdafuncties bij containers en algoritmes

In dit hoofdstuk wordt getoond hoe je lambdafuncties kan inzetten bij STL containers en algoritmes. Dit is geen compleet overzicht van alle STL algoritmes die er zijn; het aantal is te groot om van elk een voorbeeld op te nemen in deze cursustekst.

De volgende pagina's geven een goed overzicht van alle STL algoritmes.

- http://www.cplusplus.com/reference/algorithm/
- http://en.cppreference.com/w/cpp/algorithm

We geven nu een aantal voorbeelden die tonen hoe je lambdafuncties kan inzetten bij algoritmes toegepast op containers.

Dit voorbeeld sorteert een `vector` in omgekeerde volgorde.

```
vector<int> lijst2 = { 5, 6, 1, 9, 7};
sort(lijst2.begin(), lijst2.end(),
     [](const int &a, const int &b) -> bool { return a > b; });
```

De lijst bestaat uit `int` waarden en daarom zijn de parameters van de lambdafunctie ook `int`. Voor een betere efficiëntie is er voor de parameters gekozen voor het referentietype en om te vermijden dat de parameters gewijzigd worden is er `const` gebruikt. Het doel van deze lambdafunctie is een `true` terug te geven als de 2 elementen in dalende volgorde staan. Dit zorgt ervoor dat de lijst in dalende volgorde wordt gesorteerd.

Het volgende voorbeeld gaat na of alle getallen in de lijst positief zijn. Ook hier wordt de lambdafunctie gebruikt om de voorwaarde te bepalen.

```
array<int,8> tab = {2,5,6,11,16,17,19,24};

if (all_of(tab.begin(), tab.end(), [](int i){return i >= 0;}) )
{
      std::cout << "alle getallen zijn positief\n";
}
```

Het volgende voorbeeld zoekt het eerste negatieve getal. Hiervoor wordt het `find_if` algoritme gebruikt. Dit geeft een iterator terug die, in dit geval, naar het eerst negatieve getal wijst. Indien er geen negatief getal gevonden wordt, dan wordt de iterator `tab.end()` teruggegeven.

```
vector<int> tab = {3, 5, 6, -4, 8, 1, 5};

vector<int>::iterator it = find_if (tab.begin(), tab.end(),
             [](int i)->bool{ return i < 0;});
std::cout << "Het eerste negatieve getal is " << *it << '\n';
```

Het volgende voorbeeld vult een array met een oplopende reeks gehele getallen. Een `array` is te vergelijken met een `vector` maar heeft wel een vaste grootte. In de lambdafunctie wordt de lokale variabele `teller` gebruikt, deze teller moet zijn waarde behouden over de oproepen van de lambdafunctie. Daarom staat de declaratie van `teller` buiten de lambdafunctie. Hierdoor is een capture nodig. Omdat de lambdafunctie `teller` wijzigt, moet de capture een ampersand bevatten.

```
array<int, 10> tab;

int teller = 0;
generate(tab.begin(), tab.end(), [&teller](){ return teller++;});
```

Het volgende voorbeeld sommeert alle getallen in een reeks. De lambdafunctie krijgt via de parameter `a` de voorlopige som van alle getallen, de parameter `b` is het volgende op te tellen getal. De som van beide getallen wordt teruggegeven. Het algoritme `accumulate` krijgt vier parameters: het begin en einde van de reeks, de startwaarde voor het sommeren, in dit geval `0`, en de lambdafunctie die de sommeerbewerking uitvoert.

```
vector<int> tab = { 3, 5, 7, 8, 9};

int totaal = accumulate(tab.begin(), tab.end(), 0, [](const int &a, const int &b) -> int { return a +
```

Het volgende voorbeeld doet ongeveer hetzelfde zoals het vorige voorbeeld, namelijk sommeren, maar deze keer is elk element in de reeks een pointer naar een `Punt` object. Vermits in deze cursustekst als programmeerstijl voor het beheer van objecten pointers naar objecten worden gebruikt, is in dit voorbeeld getracht om de reeks als `vector<Punt *>` te declareren. Dit heeft een aantal gevolgen die zo dadelijk duidelijk gemaakt worden. Nu volgt de volledig broncode van het voorbeeld.

```
#include <iostream>
#include <vector>
#include <algorithm>

class Punt
{
public:
   int x;
   int y;

   Punt(int ix, int iy): x(ix), y(iy)
   {
   }
};


int main()
{
   vector<Punt *> tab = {new Punt(3,4), new Punt(5,6), new Punt(7,8)};

   Punt *totaal = accumulate(tab.begin(), tab.end(),
                    new Punt(0,0),
                    [](const Punt *a, const Punt *b) -> Punt * { return new Punt(a->x + b->x, a->y +

   std::cout << totaal->x << " " << totaal->y << std::endl;

   return 0;
}
```

Eerst wordt de klasse `Punt` vastgelegd.

```
class Punt
{
public:
   int x;
   int y;

   Punt(int ix, int iy): x(ix), y(iy)
   {
   }
```

```
};
```

Voor het gemak zijn de datamembers `x` en `y` `public` en zijn behalve de constructor geen methoden.

In `main()` wordt eerst de `vector` met `Punt` objecten gevuld. Dit gebeurt met een initialisatielijst tussen accoladen, deze schrijfwijze in recent toegevoegd aan C++.

```
vector<Punt *> tab = {new Punt(3,4), new Punt(5,6), new Punt(7,8)};
```

Om de punten te maken heb je uiteraard `new()` nodig.

Bij de oproep van `accumulate()` zijn de eerste twee parameters het begin en einde van de reeks. De derde parameter is de startwaarde voor de sommering. Hier kan je niet anders dan een `Punt *` waarde door te geven. Daarom wordt een vers object `new Punt(0,0)` doorgegeven.

```
   Punt *totaal = accumulate(tab.begin(), tab.end(),
                     new Punt(0,0), ... );
```

Het resultaat van het algoritme is ook van het `Punt *` type. Je krijgt een `Punt` object terug waarin de totale sommen staan.

De lambdafunctie krijgt als parameter twee `Punt` objecten en berekent de volgende partiële som in `x` en in `y` en maakt een nieuw object om dit tijdelijke resultaat terug te geven.

```
[](const Punt *a, const Punt *b) -> Punt * { return new Punt(a->x + b->x, a->y + b->y); });
```

Dit is dan de uitleg van het voorbeeld. Als resultaat krijg je de getallen `3+5+7` en `4+6+8`.

Nu is er wel een aardig probleem met dit pointervoorbeeld. Als je dit voorbeeld test met `valgrind` zal je merken dat geen enkel object vrijgegeven wordt. Dit probleem moeten we oplossen.

Het volgende voorbeeld is de verbetering. Hierin zijn de nodige `delete` bewerkingen bijgevoegd.

```
int main()
{
   vector<Punt *> tab = { new Punt(3,4), new Punt(5,6), new Punt(7,8)};

   Punt *totaal = accumulate(tab.begin(), tab.end(),
                     new Punt(0,0),
                     [](const Punt *a, const Punt *b) -> Punt *
                  {
                     Punt *nieuw = new Punt(a->x + b->x, a->y + b->y);
                     delete a;
                     return nieuw;
                  });

   std::cout << totaal->x << " " << totaal->y << std::endl;

   delete totaal;
   for (Punt *p: tab)
   {
      delete p;
   }
   tab.clear();

   return 0;
}
```

Dit zijn de noodzakelijke `delete`'s:

- Heel de reeks wordt vrijgegeven met een `for`.
- In de lambdafunctie wordt het `a` object vrijgegeven omdat er een ander tijdelijk object in de plaats komt voor het tussenresultaat. Eerst wordt het nieuwe object gemaakt, dan wordt `a` vrijgegeven en dan wordt het nieuwe object teruggegeven.
- Uiteindelijk wordt het `totaal` object ook vrijgegeven.

Deze oplossing vraagt wel puzzlewerk om de vrijgave van de objecten juist te krijgen. Een andere oplossing is het gebruik van het STL type `shared_ptr` Dit is een recente toevoeging die deel uitmaakt van C++. Hiermee kan je objectenpointers gebruiken zonder de objecten expliciet te moeten vrijgeven. Dit lijkt op de garbage collection van Java.

```
#include <iostream>
#include <memory>
#include <vector>
#include <algorithm>

class Punt
{
public:
   int x;
   int y;

   Punt(int ix, int iy): x(ix), y(iy)
   {
   }
};

using Punt_p =  std::shared_ptr<Punt>;

int main()
{
   Punt_p pp = std::make_shared<Punt>(5,6);
   std::vector<Punt_p> tab =
   {
      std::make_shared<Punt>(3,4),
      std::make_shared<Punt>(5,6),
      std::make_shared<Punt>(7,8)
   };

   Punt_p totaal = accumulate(tab.begin(), tab.end(),
                    std::make_shared<Punt>(0,0),
                    [](const Punt_p a, const Punt_p b) -> Punt_p
                  {
                     Punt_p nieuw = std::make_shared<Punt>(a->x + b->x, a->y + b->y);
                     //delete a;
                     return nieuw;
                  });

   std::cout << totaal->x << " " << totaal->y << std::endl;

   //delete totaal;
   for (Punt_p p: tab)
   {
      //delete p;
   }
   tab.clear();

   return 0;
}
```

Deze sectie is zeker niet compleet; niet alle algoritmes zijn hier met een voorbeeld beschreven. Je kan altijd de bovenstaande links gebruiken om verdere voorbeelden te zoeken.

Lambdafuncties met expliciete capture

In C++14 bestaat de mogelijkheid om expliciet aan te geven hoe variabelen in een capture gekoppeld worden. We geven een demonstratie van deze techniek met een voorbeeld. Het voorbeeld toont ook hoe de klasse van het functieobject eruit ziet dat het gedrag van de capture simuleert. Het is zo dat de compiler voor elke lambdafunctie met capture een nieuwe klasse genereert dit dienst doet als functieobject[35].

De volledige broncode van het voorbeeld volgt nu:

```
#include <iostream>

class Capture
{
private:
   int &xx;

public:
   Capture(int &xxx) : xx(xxx)
   {
   }

   void operator()(int y)
   {
      xx += y;
   }
};

int main()
{
   int x = 5;

   auto fu = [&xx = x](int y)
   {
      xx += y;
   };

   fu(2);
   std::cout << "x " << x << std::endl;

   Capture c(x);
   c(2);
   std::cout << "x " << x << std::endl;
}
```

Als we een variabele willen verhogen met een lambdafunctie, dan doen we dat zo:

```
int x = 5;

auto fu = [&xx = x](int y)
{
   xx += y;
};

fu(2);  // verhoog x met 2
std::cout << "x " << x << std::endl;
```

De variabele `x` start met de waarde `5`. De lambdafunctie `fu` heeft een capture die als `[&xx = x]` geschreven wordt. Dit is een nieuwe C++14 syntax. Hier wordt gesteld dat de variabele `x` van buiten de lambdafunctie binnen de lambdafunctie bekend zal zijn onder de naam `xx`. En de ampersand geeft aan dat de variabele `x` vanuit de lambdafunctie kan gewijzigd worden. Merk op dat de variabele `x` absoluut moet blijven bestaan

[35] Een `functieobject` is een object van een klasse met enkel een `operator()()` als enige methode.

tot en met het moment dat de lambdafunctie gestart wordt. Als dat niet zo is `xx` zijn backing variabele kwijt en gebeuren er onvoorspelbare dingen.

De klasse `Capture` simuleert exact het gedrag van de lambdafunctie en is daarom interessant om lambdafuncties te begrijpen. De datamember `xx` houdt een referentie bij naar `x` (ook in dit geval moet `x` blijven bestaan). Door de oproep van `c(2)` wordt de enige methode gestart.

```
Capture c(x);
c(2);
std::cout << "x " << x << std::endl;
```

Het is dus de datamember die het mogelijk maakt om de waarde in de capture te onthouden. En als we captures gebruiken, is het belangrijk om geen geheugenfouten te krijgen door gebroken referenties. Verder is het zo dat de expliciete capture duidelijk aangeeft wat de namen van de betrokken variabelen zijn. In de notatie `[&xx = x]` is `x` de variabele die gevangen wordt en is `xx` de variabele waarmee de gevangen variabele kan bereikt worden. Het is zelfs mogelijk om aan de rechterzijde van het `=` teken een functieoproep te plaatsten. Dus dit mag geschreven worden:

```
auto fu = [n = doe()](int y)
{
   ...
};
```

Multithreading

Uiteraard was het al langer mogelijk om met C en C++ multitasking in te bouwen in programma's maar altijd waren hiervoor API's nodig die niet gestandaardiseerd waren. De bekendste voorbeelden zijn `fork()` (enkel in Linux) en de *pthread* bibliotheek. Deze laatste is goed beschikbaar voor alle platformen maar is wel een C bibliotheek. Voor C++ is er nu met de komst van C++11/C++14 wel een standaard voor multitasking in C++.

Hier zijn een aantal voorbeelden gebaseerd op de volgende labotekst.

- https://www.classes.cs.uchicago.edu/archive/2013/spring/12300-1/labs/lab6/

`vb1.cpp`, een eenvoudige thread

In dit voorbeeld wordt een thread gestart. Met `join()` kan je op het einde van de thread wachten. De naam van de uit te voeren functie wordt als een parameter aan de constructor van `std::thread` meegegeven. In C en ook C++ doet de naam van de functie dienst als adres van de functie.

Broncode: vb1.cpp

```
#include <iostream>
#include <thread>

void func(int x)
{
   std::cout << "in thread " << x << std::endl;
}

int main()
{
   std::thread th(func, 100);
   th.join();

   std::cout << "thread afgelopen\n";
   return 0;
}
```

Denk eraan dat een thread een lichtgewicht proces is. Alle threads hebben toegang tot hetzelfde gemeenschappelijke geheugen. Deze eigenschap maakt het opstarten van een nieuwe thread relatief

gemakkelijk maar het nadeel is dan wel dat er race- en andere synchronisatieproblemen kunnen optreden als meerdere threads toegang hebben tot gemeenschappelijke bronnen, in dit geval het geheugen.

vb2.cpp, gemeenschappelijk geheugen schrijven

Hier worden meerdere threads gestart die elk een globale variabele wijzigen. Dit is een gevaarlijk situatie omdat er race-effecten kunnen ontstaan waardoor het eindresultaat fout kan zijn.

Al de threads worden in een `vector` opgeslagen zodat achteraf een `join()` op elke thread kan uitgevoerd worden.

Broncode: vb2.cpp

```
#include <iostream>
#include <vector>
#include <thread>

int accum = 0;

void square(int x)
{
   std::cout << "in thread " << x << std::endl;
   accum += x*x;
}

int main()
{
   std::vector<std::thread> ths;
   for (int i=1; i<=20; i++)
   {
      ths.push_back(std::thread(square, i));
   }

   for (auto &th: ths)
   {
      th.join();
      std::cout << "thread afgelopen\n";
   }

   std::cout << "accum = " << accum << std::endl;
   return 0;
}
```

Opnieuw wordt in dit voorbeeld een bestaande functie `square(int x)` als thread gestart. Deze functie krijgt één parameter `x` die bij de start van de thread wordt doorgegeven. Elke thread berekent het kwadraat van de doorgegeven parameter. Dit resultaat wordt bij de globale variabele `accum` opgeteld. Als deze verhoging tegelijkertijd door meerdere threads wordt uitgevoerd, kunnen er problemen ontstaan; het eindresultaat zal dan niet correct zijn.

vb3.cpp, thread en mutex

Dit is de vorige versie uitgebreid met een *mutex*. Hierdoor wordt het race-effect vermeden. Elke thread doet eerst een `lock()` voordat de globale variabele gewijzigd wordt. Achteraf gebeurt dan een `unlock()`.

Broncode: vb3.cpp

```
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>

int accum = 0;
```

```
std::mutex accum_mutex;

void square(int x)
{
   int temp = x*x;
   accum_mutex.lock();
   accum += temp;
   accum_mutex.unlock();
}

int main()
{
   std::vector<std::thread> ths;
   for (int i=1; i<=20; i++)
   {
      ths.push_back(std::thread(square, i));
   }

   for (auto &th: ths)
   {
      th.join();
      std::cout << "thread afgelopen\n";
   }

   std::cout << "accum = " << accum << std::endl;
   return 0;
}
```

Een mutex heeft twee interne toestanden: vrij of bezet. Met `lock()` geet de toestand van vrij naar bezet op voorwaarde dat de toestand vrij is. Als bij de oproep van `lock()` de toestand bezet is, moet de oproepende thread wachten tot de toestand vrij wordt. Dit zorgt ervoor dat slechts één thread per keer doorheen de `lock()` oproep geraakt. Een thread kan op deze manier een mutex bezet houden. Met `unlock()` wordt de mutex opnieuw vrij gemaakt en komt een andere thread aan de beurt. Het gevolg is dat er een wederzijdse uitsluiting ontstaat voor het gebied tussen `lock()` en `unlock()`. Om de werking efficiënter te krijgen is de berekening van het kwadraat buiten het bescshermde gebied geplaatst. Hierdoor is de bezettingstijd zo kort mogelijk.

vb4.cpp, thread en atomic

Dit voorbeeld levert een andere oplossing voor hetzelfde probleem. De oplossing werkt niet met een mutex maar wel met een *atomic*. Ook hier is de toegang beveiligd met wederzijdse uitsluiting.

Broncode: vb4.cpp

```
#include <iostream>
#include <vector>
#include <thread>
#include <atomic>

std::atomic<int> accum(0);

void square(int x)
{
   std::cout << "in thread " << x << std::endl;
   accum += x*x;
}

int main()
{
   std::vector<std::thread> ths;
```

```
   for (int i=1; i<=20; i++)
   {
      ths.push_back(std::thread(square, i));
   }

   for (auto &th: ths)
   {
      th.join();
      std::cout << "thread afgelopen\n";
   }

   std::cout << "accum = " << accum << std::endl;
   return 0;
}
```

Bij deze oplossing zit de wederzijdse uitsluiting ingebouwd in de `operator+=()` bewerking van de `atomic` klasse. Het voordeel van deze oplossing dat enkel de declaratie van `accum` moet aangepast worden.

vb5.cpp, async

Dit voorbeeld demonstreert het gebruik van `async()`. Hiermee kan je een bewerking, die een resultaat moet geven als een thread starten. Threads maken is niet zo moeilijk maar resulaten van threads teruggeven is wel lastig; `async` biedt hiervoor een oplossing.

In de `main()` staan twee oproepen:

- De eerste staat in commentaar, hier is het niet zeker of er wel een aparte thread gestart wordt.
- De tweede oproep heeft als parameter een constante `launch::async` die aangeeft dat er meteen een thread moet gestart worden.

Met `get()` kan je het resultaat van de berekening opvragen.

Broncode: vb5.cpp

```
#include <iostream>
#include <future>
#include <chrono>

int square(int x)
{
   std::this_thread::sleep_for(std::chrono::milliseconds(100));
   return x*x;
}

int main()
{
   //auto a = std::async(square, 10);
   auto a = std::async(std::launch::async, square, 10);
   int v = a.get();

   std::cout << "v = " << v << std::endl;
   return 0;
}
```

Met de oproep die in commentaar staat, wordt de `square()` functie gestart maar het is niet zeker of die functie binnen een thread uitgevoerd wordt. Met de tweede oproep wordt er wel een thread gemaakt. Als de berekening in de thread lang duurt, is het waarschijnlijk dat de oproep van `get()` al start voordat het resultaat in de thread teruggegeven wordt. In dat geval zal `get()` blokkeren en wachten tot het resultaat er is.

vb6.cpp, conditievariabele

Door een conditievariabele in te zetten kan een thread wachten op een bepaalde voorwaarde. In het voorbeeld wacht de reporter thread tot een waarde in `value` is ingesteld.

De assigner thread stelt de waarde in.

Door het gebruik van de conditievariabele krijg je wel polling maar deze pollingtechniek reageert snel op een wijziging van de voorwaarde zonder dat er veel CPU verbruik is.

Broncode: vb6.cpp

```
#include <iostream>
#include <thread>
#include <condition_variable>
#include <mutex>
#include <queue>

std::condition_variable cond_var;
std::mutex m;

int main()
{
   int value = 10;
   bool notified = false;

   std::thread reporter([&]()
   {
      std::unique_lock<std::mutex> lock(m);
      while (!notified)
      {
          cond_var.wait(lock);
      }
      std::cout << "value is " << value << std::endl;
   });

   std::thread assigner([&]()
   {
      value = 20;
      notified = true;
      cond_var.notify_one();
   });

   reporter.join();
   assigner.join();
   return 0;
}
```

Het bovenstaande voorbeeld kent een aantal nieuwigheden. Het eerst dat opvalt is het gebruik van lambdafuncties bij de start van de threads. Je maakt een `std:thread` variabele die met een lambdafunctie wordt geïnitialiseerd. Beide lambdafuncties hebben `[&]` als capture. Dit betekent dat ze toegang hebben tot alle lokale variabelen van `main()`. De mutex met naam `lock` zorgt voor de wederzijdse uitsluiting. Het wijzigen van `value` en `notified` vanuit de threads is dus geen probleem. Naast de mutex is er nog een tweede variabele die betrokken is in de synchronisatie. Deze is van het type `std::condition_variabele`. Deze variabele organiseert het wachten op de voorwaarde. De voorwaarde staat geschreven als `while (!notified)`. Het is hier een `while` en geen `if` omdat we waarschijnlijk de voorwaarde meerdere malen moeten testen. Zolang de voorwaarde waar is blijven we wachten met `cond_var.wait(lock)`. In deze oproep moet de mutexvariabele `lock` als parameter meegegeven worden. De reden hiervoor is simpel en levensnoodzakelijk: we zijn in de `while` geraakt door eerst de mutex te reserveren. De mutex is dus bezet. Als we nu gaan wachten, zou de mutex bezet blijven en zou het hele programma (beter gezegd: alle threads in dit programma) blokkeren. Om dit te vermijden wordt bij het uitvoeren van `cond_var.wait(lock)` de

mutex meegegeven zodat die kan vrijgegeven worden. Dit betekent dan ook dat wanneer de thread uit de waittoestand komt deze mutex opnieuw moet gereserveerd worden voordat we verder gaan. Ook dit gebeurt automatisch.

We hebben nu de werking van de `reporter` thread uitgelegd. De `assigner` thread is heel wat eenvoudiger. Deze thread roept de functie `cond_var.notify_one()` op. Hiermee wordt één thread wakker geschud die in wait zit. Je zou ook `cond_var.notify_all()` kunnen gebruiken maar dat is enkel nuttig als er meerdere threads in de waittoestand zitten.

`vb7.cpp`, consumer-producer probleem

Dit is een klassiek probleem dat met een conditievariabele is opgelost. De conditievariabele en mutex maken nu deel uit van de klasse `Goods`. Ze zijn daar datamembers. Er is ook een datamember `q`, dit is de wachtrij.

Er zijn twee methoden in de klasse `goods`:

- `push()`

  Deze methode plaatst een element op de wachtrij. Er hoeft niet gewacht te worden. Met `notify_one()` wordt de andere thread gewekt.

- `pop()`

  Deze methode wacht op een element. Als de wachtrij leeg is, moet er gewacht worden. Dit gebeurt met `wait()`.

Naast de conditievariabele is er bij dit mechanisme nog een mutex. Deze mutex zit verpakt in een `lock_guard` variabele.

```
{
   std::lock_guard<std::mutex> lock(m);
   q.push(x);
}
```

Deze variabele staat in een eigen `{}` block. Hierdoor lopen automatisch de constructor en de destructor van de `lock_guard`. Deze zorgen respectievelijk voor de `lock()` en `unlock` van de mutex.

In de `main()` functie worden de producer en consumer elk als een thread gestart.

Broncode: vb7.cpp

```
#include <iostream>
#include <thread>
#include <condition_variable>
#include <mutex>
#include <queue>

class Goods
{
private:
   std::condition_variable cond_var;
   std::mutex              m;
   std::queue<int>         q;

public:
   Goods()
   {
   }
   void push(int x)
   {
      {
         std::lock_guard<std::mutex> lock(m);
         q.push(x);
```

```
      }
      cond_var.notify_one();
   }
   int pop()
   {
      std::unique_lock<std::mutex> lock(m);
      while (q.empty())
      {
         cond_var.wait(lock);
      }
      int v = q.front();
      q.pop();
      return v;
   }
};

int main()
{
   int n = 20;
   Goods goods;

   std::thread producer([&]()
   {
      for (int i=0; i<n; i++)
      {
         std::cout << "i push " << i << std::endl;
         goods.push(i);
      }
   });

   std::thread consumer([&]()
   {
      for (int i=0; i<n; i++)
      {
         int v = goods.pop();
         std::cout << "v pop " << v << std::endl;
      }
   });

   producer.join();
   consumer.join();
   return 0;
}
```

Door gebruik te maken van een klasse krijgen we een inkapseling van de synchronistatie. Dit maakt het geheel overzichtelijker; de globale variabelen zijn nu datamembers geworden.

vb8.cpp promise-future

In dit voorbeeld worden *promise* en *future* gedemonstreerd. Deze constructie kan je gebruiken om een resultaat van een asynchroon lopende bewerking op te vragen. Het resultaat wordt na de berekening met `set_value()` in de promise geplaatst. De promise is een globale variabele die als een parameter aan de lambdafunctie wordt meegegeven.

Met `get_future()` wordt eerst een future opgehaald en in deze future kan met `get()` het uiteindelijke resultaat opgevraagd worden.

Broncode: vb8.cpp

```
#include <iostream>
```

```
#include <future>

int main()
{
   std::promise<int> prms;

   auto th = std::thread(
      [](std::promise<int> &prms, int x) -> int
      {
         prms.set_value(2*x);
      },
      std::ref(prms), 15);

   th.join();
   auto ftr = prms.get_future();
   int r = ftr.get();

   std::cout << "r " << r << std::endl;

   return 0;
}
```

vb9.cpp async en future

In dit voorbeeld is de promise/future verborgen in een *async*. Deze async geeft meteen een future terug om het resultaat op te halen.

Broncode: vb9.cpp

```
#include <iostream>
#include <future>

int main()
{
   auto ftr = std::async(
      [](int x) -> int
      {
         return 2*x;
      },
      15);

   int r = ftr.get();
   std::cout << "r " << r << std::endl;

   return 0;
}
```

Coroutines in C++

De laatste standaard van C++ voorziet nog geen implementatie van coroutines. Daarom wordt hier de *Boost* bibliotheek gebruikt. Waarschijnlijk komen de Boost coroutines pas na C++17 standaard in C++ terecht.

Pull

Dit voorbeeld toont hoe de coroutine een reeks getallen genereert. Deze reeks wordt door `main()` overgenomen.

- coroutine_pull.cpp

```
#include <iostream>
#include <boost/coroutine2/all.hpp>

typedef boost::coroutines2::coroutine<int>   coro_t;

coro_t::pull_type source(
   [&](coro_t::push_type& sink)
   {
      int first  = 1;
      int second = 1;
      sink(first);
      sink(second);
      for(int i=0;i<8;++i)
      {
         int third= first + second;
         first  = second;
         second = third;
         sink(third);
      }
   });

int main()
{
   std::cout << "main() start\n";

   for(auto i:source)
   {
       std::cout << i <<  " ";
   }
}
```

In het bovenstaande voorbeeld is de variabele `source` gedeclareerd als een *pull* coroutine. Dat betekent dat je vanuit `main()` er getallen kan uithalen. Het is `main()` die de *pull* doet. In de typedefinitie van `coro_t` zie je `coroutine<int>`. Dit wil zeggen dat de doorgegeven data van het `int` type moet zijn. De `source` coroutine krijgt een lambdafunctie als parameter. Deze functie zal uitgevoerd worden als coroutine. De lambdafunctie zelf heeft ook een parameter. Dit is `sink`. Dit is de tegenhanger van *pull*, namelijk *push*. Via `sink` kan de coroutine `int` getallen doorsturen naar `main()`. In `main()` wordt `source` gebruikt om de doorgegeven getallen op te vangen.

Merk op dat er geen echt parallellisme is zoals bij threads. Owel loopt `main()` ofwel loopt de coroutine. Als één van beide delen een oneindige lus zou maken, komt het andere deel nooit meer aan bod. Bij threads is dat niet zo. Daarom spreekt men bij coroutines ook van coöperatieve multitasking.

Push

In dit voorbeeld wordt de richting omgekeerd: `main()` levert de woorden en de coroutine doet de verwerking ervan.

- coroutine_push.cpp

```
#include <iostream>
#include <iomanip>
#include <boost/coroutine2/all.hpp>

typedef boost::coroutines2::coroutine<std::string>   coro_t;

struct FinalEOL
{
   ~FinalEOL()
   {
```

```
      std::cout << std::endl;
   }
};

const int num=5, width=15;
coro_t::push_type writer(
   [&](coro_t::pull_type& in)
   {
      // finish the last line when we leave by whatever means
      FinalEOL eol;
      // pull values from upstream, lay them out 'num' to a line
      for (;;)
      {
         for (int i=0;i<num;++i)
         {
            // when we exhaust the input, stop
            if(!in) return;
            std::cout << std::setw(width) << in.get();
            // now that we've handled this item, advance to next
            in();
         }
         // after 'num' items, line break
         std::cout << std::endl;
      }
   });

std::vector<std::string> words
{
   "peas", "porridge", "hot", "peas",
   "porridge", "cold", "peas", "porridge",
   "in", "the", "pot", "nine",
   "days", "old"
};

int main()
{
   std::cout << "main() start\n";

   // Dit werkt
   //begin(writer) = "abc";

   // Dit werkt ook
   writer("def");
   writer("ghi");

   std::copy(begin(words),end(words),begin(writer));
}
```

Makefile

Bij de compilatie moet je `c++11` vermelden, `c++14` werkt ook.

```
all: coroutine_pull coroutine_push

coroutine_pull: coroutine_pull.o
    g++ -o coroutine_pull coroutine_pull.o -lboost_coroutine -lboost_context

coroutine_pull.o: coroutine_pull.cpp
    g++ -c -g -std=c++11 coroutine_pull.cpp
```

```
coroutine_push: coroutine_push.o
    g++ -o coroutine_push coroutine_push.o -lboost_coroutine -lboost_context

coroutine_push.o: coroutine_push.cpp
    g++ -c -g -std=c++11 coroutine_push.cpp
```

Pull2

Dit is een eenvoudiger pull voorbeeld.

- coroutine_pull2.cpp

```
#include <iostream>
#include <boost/coroutine2/all.hpp>

typedef boost::coroutines2::coroutine<int>   coro_t;

coro_t::pull_type source( // de constructor start de coroutinefunctie
   [&](coro_t::push_type& sink)
   {
      sink(1); // geef {1} door aan de maincontext
      sink(1);
      sink(2);
      sink(3);
      sink(5);
      sink(8);
   });

int main()
{
   while(source) // ga na of de pull-coroutine nog bestaat
   {
      int ret = source.get(); // haal de data
      std::cout << ret <<  " ";
      source();              // context-switch naar coroutinefunctie
   }
}
```

Push2

Dit is een eenvoudiger push voorbeeld.

- coroutine_push2.cpp

```
#include <iostream>
#include <iomanip>
#include <boost/coroutine2/all.hpp>

typedef boost::coroutines2::coroutine<int>   coro_t;

coro_t::push_type sink( // de constructor start de coroutinefunction NIET
   [&](coro_t::pull_type& source)
   {
      for (int i:source)
      {
         std::cout << i <<  " ";
      }
   });

int main()
```

```
{
   std::vector<int> v{1,1,2,3,5,8,13,21,34,55};

   for(int i: v)
   {
      sink(i); // push {i} naar de coroutinefunctie
   }
}
```

Push som

En dit is de labooefening. Stuur eerst het aantal door en daarna de getallen volgens het aantal. De coroutine berekent de som.

- coroutine_push_som.cpp

```
#include <iostream>
#include <iomanip>
#include <boost/coroutine2/all.hpp>

typedef boost::coroutines2::coroutine<int> coro_t;

coro_t::push_type sink(
   [&](coro_t::pull_type& source)
   {
      int aantal = source.get();
      std::cout << "aantal is " << aantal << "\n";
      source();

      int som = 0;
      for (int i = 0; i<aantal; i++)
      {
         int g = source.get();
         std::cout << "g is " << g << "\n";
         som  += g;
         source();
      }
      std::cout << "de som is " << som << "\n";
   });

int main()
{
   sink(4); // het aantal getallen is 4

   sink(1); // 1ste getal
   sink(2); // 2de getal
   sink(3); // 3de getal
   sink(4); // 4de getal
   std::cout << "einde\n";
}
```

Link

- http://www.boost.org/doc/libs/1_63_0/libs/coroutine2/doc/html/coroutine2/coroutine/asymmetric.html

De Qt toolkit www.qt.io is een C++ bibliotheek waarmee je grafische applicaties kan ontwerpen. Ze is de basis waarmee de Linux KDE vensteromgeving is ontworpen. Qt bestaat al sinds 1992 en is tegenwoordig taal en platformonafhankelijk. Oorspronkelijk was Qt, die zelf in C++ is geschreven, enkel bedoeld om hiermee grafische C++ programma's te maken maar ondertussen zijn er ook koppelingen voor andere

talen zoals Python en Java. Qt kan je niet alleen maar inzetten voor Linux GUI programma's maar ook voor Window en IOS. De multi-platform en multi-taal eigenschappen maken Qt aantrekkelijk en bijgevolg redelijk populair.

Qt is een toolkit die momenteel aan populariteit wint in de embedded systemen. Zo zijn er een aantal bedrijven die Qt inzetten in de automobielsector. Ook pretendeert Qt een oplossing te bieden voor het probleem dat je apps voor elk nieuw platform opnieuw moet programmeren. De tijd zal uitwijzen of Qt deze verwachting kan inlossen.

Voor dit vak is Qt wel een voor de hand liggende keuze als GUI toolkit. Vermits de oefeningen in een Linux omgeving worden gemaakt en dat Qt er standaard is geïnstalleerd, maken deze keuze wel onvermijdelijk.

Het kleinste Qt voorbeeld

Dit is een klein Qt voorbeeld met een `paint()` functie. Dit betekent dat het programma zelf kan beslissen wat er in een canvas moet getekend worden. Voor de oefeningen is dit het programma dat als basis dient voor de oefening in het labo. In het programma heb je maar één klasse en een `main()` functie. Deze laatste zorgt voor het opstarten van het programma.

Dit is de volledige broncode:

```
#include <QtGui>

class MyWidget : public QWidget
{
public:
   MyWidget(QWidget *parent = 0);

protected:
   virtual void paintEvent(QPaintEvent *event);
};


MyWidget::MyWidget(QWidget *parent)
     : QWidget(parent)
{
     setWindowTitle("MyWidget");
     resize(200, 200);
}

void MyWidget::paintEvent(QPaintEvent *event)
{
   QPainter painter(this);

   // teken een lijn
   painter.drawLine(0, 0, 100, 150);

   // teken een driehoek
   QPolygon polygon(3);
   polygon.putPoints(0, 3, 40,40, 100,40, 100,100);

   painter.setBrush(QColor(200, 100, 40));
   //painter.setBrush(Qt::cyan);

   painter.setPen(Qt::NoPen);
   painter.drawPolygon(polygon);
}

int main(int argc, char *argv[])
{
```

```
   QApplication app(argc, argv);
   MyWidget window;
   window.resize(320, 240);
   window.setWindowTitle(QApplication::translate("childwidget", "Child widget"));
   window.show();

   return app.exec();
}
```

De enige klasse in dit voorbeeld is `MyWidget`. Deze klasse is afgeleid van `QWidget`. Hiermee erft de klasse alle eigenschappen zodat het venster kan getekend worden. Er is een include van `QtGui` nodig om de `QWidget` klassen te kunnen gebruiken.

De klasse `MyWidget` heeft geen datamembers, één constructor en de `paintEvent()` methode. Deze laatste wordt automatisch opgeroepen wanneer de inhoud van het venster ongeldig is en bijgevolg opnieuw getekend moet worden.

```
#include <QtGui>

class MyWidget : public QWidget
{
public:
   MyWidget(QWidget *parent = 0);

protected:
   virtual void paintEvent(QPaintEvent *event);
};
```

De constructor stelt de titel en de grootte van het venster in.

```
MyWidget::MyWidget(QWidget *parent)
     : QWidget(parent)
{
     setWindowTitle("MyWidget");
     resize(200, 200);
}
```

De `maintEvent()` methode verzorgt het tekenwerk. Om te kunnen tekenen heb je een `QPainter` object nodig. Dit object fungeert als grafische context voor het gebied waarin getekend wordt. Hier is dat de binnenzijde van een venster. Het is de `QPainter` klasse die een hele reeks methoden ter beschikking stelt voor het tekenwerk.

```
void MyWidget::paintEvent(QPaintEvent *event)
{
   QPainter painter(this);

   // teken een lijn
   painter.drawLine(0, 0, 100, 150);

   // teken een driehoek
   QPolygon polygon(3);
   polygon.putPoints(0, 3, 40,40, 100,40, 100,100);

   painter.setBrush(QColor(200, 100, 40));
   //painter.setBrush(Qt::cyan);

   painter.setPen(Qt::NoPen);
   painter.drawPolygon(polygon);
}
```

Eerst wordt een lijn getekend en daarna een gevulde driehoek. Voor het laatste tekenwerk moet je eerst een `QPolygon` maken waarin je 3 punten van de hoeken opslaat. Daarna wordt een borstelkleur

ingesteld. Hiermee wordt de vulkleur bedoeld. En er wordt geen rand (`Qt::NoPen`) getekend. Tot slot wordt `drawPolygon()` opgeroepen die het eigenlijke tekenwerk doet.

De `main()` functie maakt een `QApplication` object en een `MyWidget` object. Dit laatste wordt getoond met `window.show()`. De applicatie wacht met `app.exec()` tot het venster gesloten wordt. een

```
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   MyWidget window;
   window.show();

   return app.exec();
}
```

Het programma compileren doe je in drie stappen:

```
qmake -project
qmake
make
```

De eerste stap genereert een `.pro` configuratiebestand met als naam de naam van de map waarin het project staat. De tweede stap genereert de `Makefile` en met de derde stap wordt de eigenlijke compilatie gestart. Door `qmake` te gebruiken hoef je zelf geen `Makefile` te maken. Als naam voor de gecompileerde applicatie wordt de naam van de map genomen.

Figuren tonen in een Qt voorbeeld

In dit voorbeeld wordt getoond hoe je een C++ model kan integreren in een Qt programma. Het model wordt in het volgende UML diagramma weergegeven:

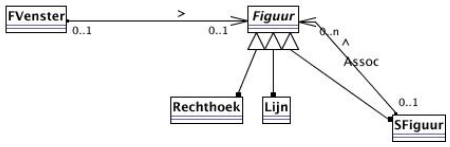

Figure 22: UML Diagramma van het Figuren voorbeeld

Gemakkelijkheidshalve staan alle klassen in hetzelfde `figuren.cpp` bestand. Dit is de volledige broncode:

```
//  Qt figuren vb

#include <vector>
#include <QtGui>

class Figuur
{
public:
   Figuur(int ix, int iy) : x(ix), y(iy)
   {
   }
   virtual void teken(QPainter *p, int dx, int dy) = 0;

protected:
   int x;
   int y;
};

class Lijn : public Figuur
{
```

```
public:
   Lijn(int ix1, int iy1, int ix2, int iy2) : Figuur(ix1, iy1), x2(ix2), y2(iy2)
   {
   }

   void teken(QPainter *p, int dx, int dy);

private:
   int x2;
   int y2;
};

class Rechthoek : public Figuur
{
public:
   Rechthoek(int ix1, int iy1, int ix2, int iy2) : Figuur(ix1, iy1), x2(ix2), y2(iy2)
   {
   }

   void teken(QPainter *p, int dx, int dy);

private:
   int x2;
   int y2;
};

class SFiguur : public Figuur
{
public:
   SFiguur(int ix, int iy) : Figuur(ix, iy)
   {
   }
   void voegbij(Figuur *f);
   void teken(QPainter *p, int dx, int dy);

private:
   vector<Figuur *> figuren;
};



void SFiguur::voegbij(Figuur *f)
{
   figuren.push_back(f);
}

void Lijn::teken(QPainter *p, int dx, int dy)
{
   p->drawLine(dx + x, dy + y, dx + x2, dy + y2);
}

void Rechthoek::teken(QPainter *p, int dx, int dy)
{
   p->drawRect(dx + x, dy + y, x2 - x, y2 - y);
}

void SFiguur::teken(QPainter *p, int dx, int dy)
{
```

```
   for (unsigned int i= 0;i<figuren.size(); i++)
   {
      figuren[i]->teken(p, dx + x, dy + y);
   }
}

class FVenster : public QWidget
{
public:
   FVenster(QWidget *parent=0);

protected:
   void paintEvent(QPaintEvent *);

private:
   SFiguur *fig;
};


FVenster::FVenster(QWidget *parent)
        : QWidget(parent)
{
   setWindowTitle("Figuren");
   setMinimumSize( 400, 300 );
   setMaximumSize( 400, 300 );

   // maak een SFiguur
   fig = new SFiguur(100, 100);
   fig->voegbij(new Rechthoek(0, 30, 30, 60));
   fig->voegbij(new Lijn(0, 30, 15, 0));
   fig->voegbij(new Lijn(15, 0, 30, 30));
}

void FVenster::paintEvent(QPaintEvent *)
{
   QPainter p(this);

   fig->teken(&p, 0, 0);
}


int main(int argc, char **argv)
{
   QApplication a( argc, argv );

   FVenster w;
   w.setGeometry( 100, 100, 200, 120 );
   w.show();
   return a.exec();
}
```

Hier volgt de uitleg van het programma.

Eerst komen de klassen van het datamodel aan bod. De klasse `Figuur` doet dienst als abstracte basisklasse. Aan de `teken()` methode zie je dat de klasse abstract is. Door de `= 0` wordt er aangegeven dat deze methode pas implementaties krijgt in de afleidingen. In elke afleiding komt een andere implementatie van de `teken()` methode. Dit is noodzakelijk omdat elke afgeleide figuur op een andere wijze moet kunnen getekend worden.

`Figuur` heeft twee datamembers `x` en `y`. Dit is de positie van de linkerbovenhoek van de figuur.

```
class Figuur
{
public:
   Figuur(int ix, int iy) : x(ix), y(iy)
   {
   }
   virtual void teken(QPainter *p, int dx, int dy) = 0;

protected:
   int x;
   int y;
};
```

De `Lijn` klasse is de eerste afleiding van `Figuur`. Deze klasse stelt een lijn voor.

```
class Lijn : public Figuur
{
public:
   Lijn(int ix1, int iy1, int ix2, int iy2) : Figuur(ix1, iy1), x2(ix2), y2(iy2)
   {
   }

   void teken(QPainter *p, int dx, int dy);

private:
   int x2;
   int y2;
};
```

De `Rechthoek` klasse stelt een rechthoek voor.

```
class Rechthoek : public Figuur
{
public:
   Rechthoek(int ix1, int iy1, int ix2, int iy2) : Figuur(ix1, iy1), x2(ix2), y2(iy2)
   {
   }

   void teken(QPainter *p, int dx, int dy);

private:
   int x2;
   int y2;
};
```

Met `SFiguur` krijgen we een complexere klasse. Met deze klasse kan je samenstellingen maken van andere enkelvoudige of samengestelde figuren. Om alle onderdelen van de samenstelling te kunnen bijhouden is er een `vector`. Elke element van deze lijst is een pointer naar `Figuur`. Met de `voegbij()` methode kan je figuren in de samenstelling bijvoegen.

```
class SFiguur : public Figuur
{
public:
   SFiguur(int ix, int iy) : Figuur(ix, iy)
   {
   }
   void voegbij(Figuur *f);
   void teken(QPainter *p, int dx, int dy);

private:
   vector<Figuur *> figuren;
```

```
};

void SFiguur::voegbij(Figuur *f)
{
   figuren.push_back(f);
}
```

Om te kunnen tekenen wordt er in elke afgeleide klasse van het model een `teken()` methode voorzien. Elk van deze methode tekent op zijn eigen wijze de overeenkomstige figuur. De `QPainter` wordt als parameter doorgegeven om effectief te kunnen tekenen. Voor het tekenwerk krijgen we respectievelijk `drawLine()`, `drawRect()` en een herhaling van `teken()` van de onderdelen van de samenstelling. Voor het laatste wordt een `for` herhaling gebruikt.

```
void Lijn::teken(QPainter *p, int dx, int dy)
{
   p->drawLine(dx + x, dy + y, dx + x2, dy + y2);
}

void Rechthoek::teken(QPainter *p, int dx, int dy)
{
   p->drawRect(dx + x, dy + y, x2 - x, y2 - y);
}

void SFiguur::teken(QPainter *p, int dx, int dy)
{
   for (unsigned int i= 0;i<figuren.size(); i++)
   {
      figuren[i]->teken(p, dx + x, dy + y);
   }
}
```

Dit zijn alle klassen voor het model. Nu bekijken we de klasse voor het venster, dit is `FVenster`.

```
class FVenster : public QWidget
{
public:
   FVenster(QWidget *parent=0);

protected:
   void paintEvent(QPaintEvent *);

private:
   SFiguur *fig;
};
```

Om te kunnen tekenen heeft de `FVenster` klasse een `paintEvent()` methode. Om te kunnen bijhouden wat er moet getekend worden, is er een datamember `SFiguur *fig`. Dit is de verwijzing naar het model.

In de constructor van `FVenster` worden de figuurobjecten gemaakt. Er wordt één `SFiguur` gemaakt die bestaat uit één `Rechthoek` en twee `Lijn`en. Dit is voldoende om een huisje te kunnen tekenen.

```
FVenster::FVenster(QWidget *parent)
        : QWidget(parent)
{
   setWindowTitle("Figuren");
   setMinimumSize( 400, 300 );
   setMaximumSize( 400, 300 );

   // maak een SFiguur
   fig = new SFiguur(100, 100);
   fig->voegbij(new Rechthoek(0, 30, 30, 60));
   fig->voegbij(new Lijn(0, 30, 15, 0));
```

```
   fig->voegbij(new Lijn(15, 0, 30, 30));
}
```

Merk op dat de oorsprong van het coordinatensysteem van de onderdelen in de `SFiguur` ligt. Dit heeft als voordeel dat als je de samenstelling verplaatst, ook alle onderdelen mee verschuiven.

De `paintEvent` methode delegeert het tekenwerk met een oproep van `teken` aan de figuren.

```
void FVenster::paintEvent(QPaintEvent *)
{
   QPainter p(this);

   fig->teken(&p, 0, 0);
}
```

Tenslotte is er nog de `main()` functie.

```
int main(int argc, char **argv)
{
   QApplication a( argc, argv );

   FVenster w;
   w.setGeometry( 100, 100, 200, 120 );
   w.show();
   return a.exec();
}
```

Opnieuw zijn dit de drie stappen om het voorbeeld te compileren.

```
qmake -project
qmake
make
```

Als je het voorbeeld start, zie je een huisje dat in het venster getekend wordt.

1. Een mier beweegt over een bord van vierkanten. Elke vierkant is zwart of wit van kleur. De mier kan enkel één stap naar boven, onder, link of rechts zetten naar het volgende vierkant. De mier heeft dus een oriëntatie in één van deze richtingen. Bij het verplaatsen gelden de volgende regels:

   - als de mier op een zwart vierkant staat, verandert de kleur van het vierkant naar wit en draait de mier 90 graden tegenwijzerzin en doet één stap vooruit.
   - als de mier op een wit vierkant staat, verandert de kleur van het vierkant naar zwart en draait de mier 90 graden wijzerzin en doet één stap vooruit.

   Start met een raster dat volledig wit is.

Bibliografie

Boeken

Stroustrup2000 3rd edition Bjarne Stroustrup 2000 ISBN 0-20170-073-5 Addison-Wesley Longman Publishing Co., Inc.

Meyers2005 3rd edition Scott Meyers 2005 ISBN 0321334876 Addison-Wesley Professional

CS106L Keith Schwartz 2010 Stanford University

Meyers2016 9th edition Scott Meyers 2016 ISBN 978-1-491-90399-5 O'Reilly Media